RISC-V processor architecture

Table of Contents

1. A basic RISC-V (ISA) implementation

We start with a simplified implementation that includes a subset of the core RISC-V instruction set:

- The **memory-reference instructions**: **load doubleword** (`ld`) and **store doubleword** (`sd`)
- The **arithmetic-logical** instructions **`add`** , **`sub`** , **`and`** , and **`or`**
- The conditional branch instruction: **branch if equal** (**`beq`**)

This subset does not include all the integer instructions (for example, shift, multiply, and divide are missing), nor does it include any floating-point instructions.
However, it illustrates the key principles used in creating a **datapath** and designing the control.
The implementation of the remaining instructions is similar.

In examining the implementation, we will have the opportunity to see how the **instruction set architecture** (**ISA**) determines many aspects of the implementation, and how the choice of various implementation strategies affects the clock rate and CPI for the computer.

1.1 An Overview of the Implementation

Let us look at the **core RISC-V instructions**, including the integer arithmetic-logical instructions, the memory-reference instructions, and the branch instructions.
Much of what needs to be done to implement these instructions is the same, independent of the **exact class of instruction**.
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 **`ld`** 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**.

Fortunately, for each of the **three instruction classes** (**memory-reference**, **arithmetic-logical**, and **branches**), the actions are largely the same, independent of the exact instruction.

The **simplicity and regularity of the RISC-V instruction set** simplify the implementation by making the execution of many of the instruction classes similar.

For example, **all instruction classes 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 **conditional branches for the equality test**.

After using the ALU, the actions required to complete various instruction classes differ.

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**.

For a **conditional branch instruction**, we may need to **change the next instruction address** based on the comparison; otherwise, the **PC should be incremented by four to get the address** of the subsequent instruction.

The following **Figure 1.1** shows the high-level view of a RISC-V implementation, focusing on the various functional units and their interconnection. Although this figure shows most of the flow of data through the processor, it omits two important aspects of instruction execution.

First, in several places, **Figure 1.1** shows data going to a particular unit as coming from two different sources. For example, the value written into the PC can come from one of two adders, the data written into the register file can come from either the ALU or the data memory, and the second input to the ALU can come from a

register or the immediate field of the instruction. In practice, these data lines cannot simply be wired together; we must add a logic element that chooses from among the multiple sources and steers one of those sources to its destination.

This selection is commonly done with a device called a **multiplexor**, although this device might better be called a **data selector.**
The second omission in **Figure 1.1** is that several of the units must be controlled depending on the type of instruction. For example, the data memory must read on a load and write on a store. The **register file must be written only on a load or an arithmetic-logical instruction**. And, of course, the ALU must perform one of several operations.

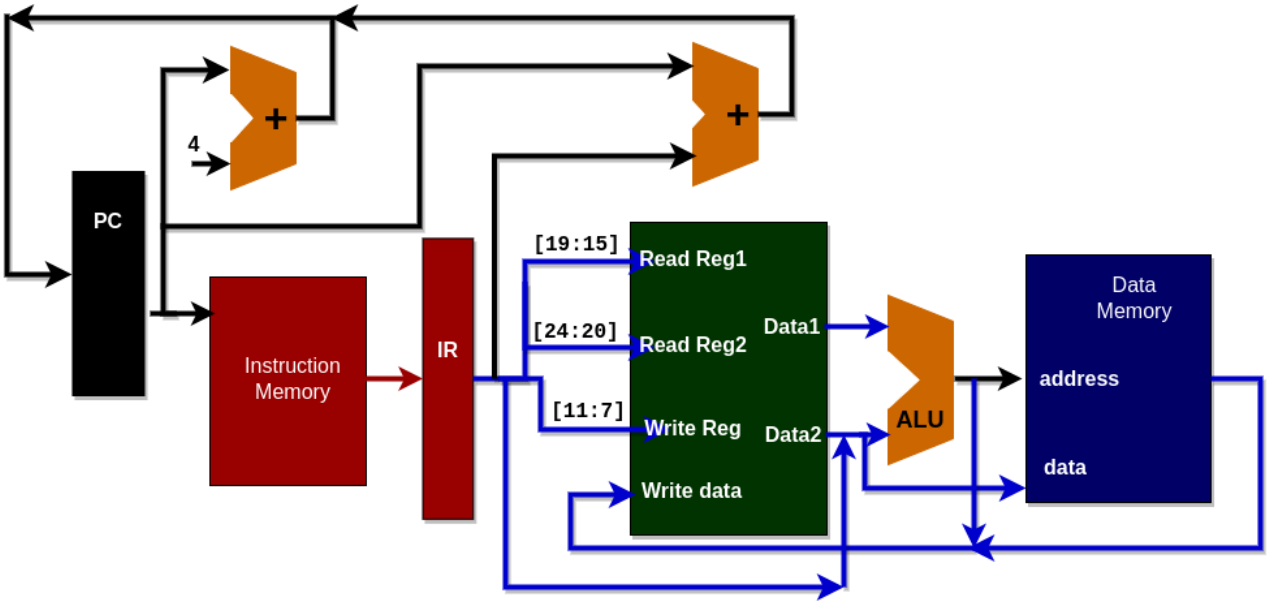


Figure 1.1 An **abstract view of the implementation** of the RISC-V subset showing the major functional units and the major connections between them.

> The **thick lines** interconnecting the functional units represent **buses**, which consist of **multiple signals**. The **arrows** are used to guide the reader in knowing how **information flows**.
>
> All instructions start by using the **program counter** to supply the instruction address to the instruction memory. After the instruction is fetched, the **register operands** used by an instruction are specified by **fields** of that instruction.
> Once the register operands have been fetched, they can be operated on to compute a memory address (for a **load** or **store**), to **compute** an arithmetic result (for an integer arithmetic-logical instruction), or an **equality check** (for a branch).
> If the instruction is an arithmetic-logical instruction, the result from the ALU **must be written to a register**. If the operation is a load or store, the **ALU result is used as an address** to either store a value from the registers or load a value from memory into the registers.
> The result from the ALU or memory is written back into the register file.
> **Branches** require the use of the **ALU output to determine** the next instruction address, which comes either from the **ALU adder** (where the PC and branch offset are summed) or from **an adder that increments** the current PC by four.

Figure 1.2 shows the **datapath** of Figure 1.1 with the **three required multiplexors** added, as well as **control lines** for the major functional units. A control unit, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors.

The top multiplexor, which determines whether **PC+4** or the branch destination address is written into the PC, is set based on the **Zero** output of the ALU, which is used to perform the comparison of a **`beq`** instruction.
The regularity and simplicity of the RISC-V instruction set mean that a simple decoding process can be used to determine how to set the control lines.

In the following Sections 1.3 and 1.4 we describe a simple implementation that uses a single long clock cycle for every instruction and follows the general form of Figures 1.1 and 1.2. In this first design, every instruction begins execution on one clock edge and completes execution on the next clock edge.

While easier to understand, this approach is not practical, since the **clock cycle must be severely stretched to accommodate the longest instruction**. After designing the control for this simple computer, we will look at **pipelined implementation** with all its complexities, including exceptions.

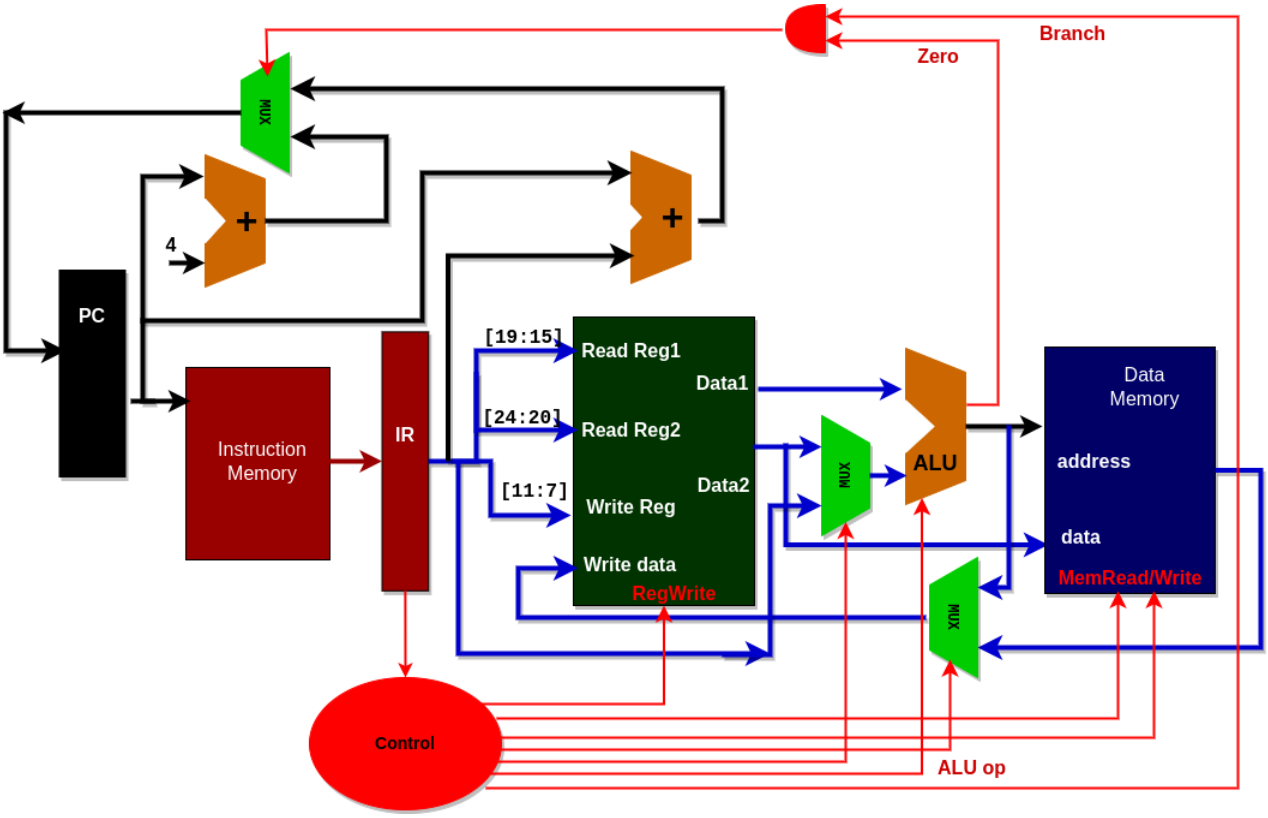


Figure 1.2 The basic implementation of the RISC-V subset, including the necessary multiplexors and control lines.

The **top multiplexor** ("MUX") controls what value replaces the `PC` (**`PC+4` or the branch destination address**); the multiplexor is controlled by the gate that "ANDs" together the Zero output of the ALU and a control signal that indicates that the instruction is a branch.
The **middle multiplexo**r, whose output **returns to the register file**, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file.
Finally, **the bottom-most multiplexor** is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store).

The added **control lines** are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.

1.2 Logic Design Conventions

To discuss the design of a computer, we must decide how the hardware logic implementing the computer will operate and how the computer is clocked. This section reviews a few key ideas in digital logic that we will use extensively in this presentation.

The **datapath** elements in the RISC-V implementation consist of **two different types of logic elements**: elements that **operate on data values** and elements that **contain state**.

The elements that operate on data values are all **combinational**, which means that **their outputs depend only on the current inputs**. Given the same input, a combinational element always produces the same output. The ALU shown in **Figure 1.1** is an example of a combinational element. Given a set of inputs, it always produces the same output because **it has no internal storage**.
Other elements in the design are not combinational, but instead **contain state**. An element contains state if it has some **internal storage**. We call these elements **state elements**.

In **Figure 4.1**, the instruction and data **memories**, as well as the **registers**, **are** all examples of **state elements**.
A state element has at least two inputs and one output. The required inputs are the **data value** to be written into the element **and the clock**, which determines **when** the data value is written. The output from a state element provides the value that was **written in an earlier clock cycle**.

For example, one of the logically **simplest state elements** is a **D-type flip-flop** , which has exactly these two inputs (a value and a clock) and one output.
In addition to flip-flops, our RISC-V implementation uses two other types of **state elements**: **memories** and **registers**, both of which appear in Figure 1.1.

The **clock is used to determine when the state element should be written**; a **state element can be read at any time**.

Logic components that contain **state** are also called **sequential**, because their outputs depend on both their inputs and the contents of the internal state. For example, the output from the functional unit representing the registers depends both on the register numbers **supplied** and on what was written into the registers **previously**.

1.2.1 Clocking Methodology

A clocking methodology defines when signals can be read and when they can be written. It is important to specify the timing of reads and writes, because if a signal is written at the same time that it is read, the value of the read could correspond to the old value, the newly written value, or even some mix of the two!

Computer designs cannot tolerate such unpredictability. A **clocking methodology is designed to make hardware predictable**.

For simplicity, we will assume an **edge-triggered clocking methodology**. An edge-triggered clocking methodology (positive edge) 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 Figure 1.3).

Because only state elements can store a data value, **any collection of combinational logic must have its inputs come from a set of state elements and its outputs written into a set of state elements**.
The inputs are values that were written in a previous clock cycle, while the outputs are values that can be used in a following clock cycle.

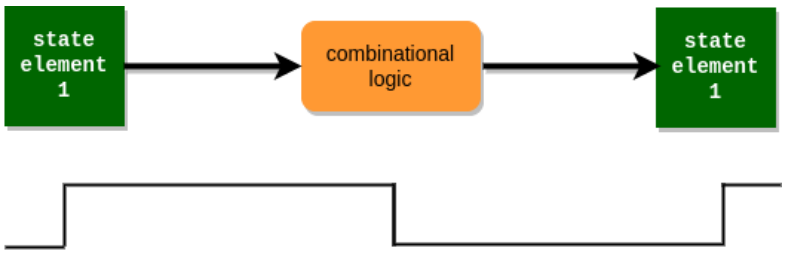


Figure 1.3 Combinational logic, state elements, and the clock are closely related.

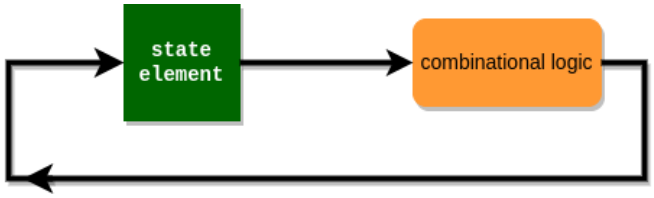
In a synchronous digital system, the clock determines when elements with state will write values into internal storage. **Any inputs to a state element must reach a stable value** (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated.
All state elements in this presentation, including memory, are assumed **positive edge-triggered**; that is, they **change on the rising clock edge**.

Figure 1.3 shows the two state elements surrounding a block of combinational logic, which operates in a single clock cycle: all signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of **one clock cycle**. The time necessary for the signals to reach state element 2 defines the length of the clock cycle.

For simplicity, we do not show a write control signal when a state element is written on every active clock edge. In contrast, **if a state element is not updated on every clock, then an explicit write control signal is required**.
Both the **clock signal and the write control signal are inputs**, and the state element is changed only when the **write control signal is asserted and a clock edge occurs**.

We use the word **asserted** to indicate a **signal** that is **logically high** and assert to specify that a signal should be driven logically high, and **deassert** or **deasserted** to represent l**ogically low.**
We use the terms assert and deassert because when we implement hardware, at times 1 represents logically high and at times it can represent logically low.
An edge-triggered methodology **allows us to read** the contents of a register, **send the value** through some combinational logic, **and write** that register **in the same clock cycle**.

Figure 1.4 gives a generic example. It doesn't matter whether we assume that all writes take place on the rising clock edge (from low to high) or on the falling clock edge (from high to low), since the inputs to the combinational logic block cannot change except on the chosen clock edge. In this book, we use the rising clock edge. With an edge-triggered timing methodology, there is no feedback within a single clock cycle, and the logic in Figure 1.4 works correctly.



Figre 1.4 An edge-triggered methodology.

An edge-triggered allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate
data values. Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. Feedback cannot occur within one clock cycle because of the edge-triggered update of the state element. If feedback were possible, this design could not work properly. Our designs in this chapter and the next rely on the edge-triggered timing methodology and on structures like the one shown in this figure.

For the **64-bit RISC-V** architecture, **nearly all of these state** and logic elements will have inputs and outputs that are **64 bits wide**, since that is the width of most of the data handled by the processor. We will make it clear whenever a unit has an input or output that is other than 64 bits in width.

The figures will indicate **buses**, which are **signals wider than 1 bit, with thicker lines**. At times, we will want to combine several buses to form a wider bus; for example, we may want to obtain a 64-bit bus by combining two 32-bit buses. In such cases, labels on the bus lines will make it clear that we are concatenating buses to form a wider bus. **Arrows are also added to help clarify the direction of the flow** of data between elements.
Finally, **color (red) indicates a control signal** contrary to a signal that carries data (blue) or address (green) ; this distinction will become clearer as we proceed through this presentation.

1.3 Building a Datapath

A reasonable way to start a datapath design is to examine the major components required to execute each class of RISC-V instructions. Let's start at the top by looking at which datapath elements each instruction needs, and then work our way down through the levels of abstraction. When we show the datapath elements, we will also show their control signals. We use abstraction in this explanation, starting from the bottom up.

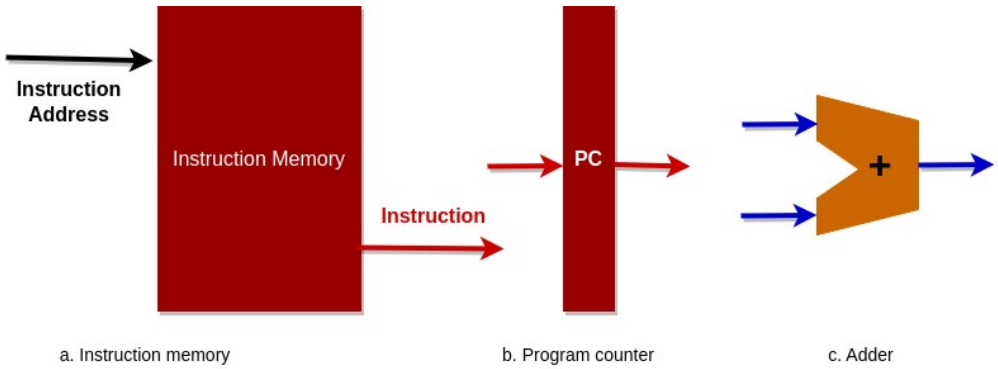


Figure 1.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.

> The **state** elements are the **instruction memory** and the **program counter**. The **instruction memory need only provide read access** because the **datapath does not write instructions**. **Since the instruction memory only reads, we treat it as combinational logic**: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. The program counter is a **64-bit register** that is written at the end of every clock cycle and thus does not need a write control signal.
> The adder is an ALU wired to always add its two 64-bit inputs and place the sum on its output.

Figure 1.5a shows the first element we need: a **memory unit to store the instructions** of a program and supply instructions given an address. **Figure 1.5b** also shows the **program counter** (`PC`), is a register that holds the address of the current instruction. Lastly, we will need an **adder to increment the PC** to the address of the next instruction.
This adder, which is combinational, can be built simply by wiring the control lines so that the control always specifies an add operation. We will draw such an ALU with the label `Add`, as in **Figure 1.5c**, to indicate that it has been permanently made an adder and cannot perform the other ALU functions.

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**.

Figure 1.6 shows how to combine the three elements from **Figure 1.5** to form a **datapath** that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

Now let's consider the **R-format** instructions (see the table below) .

Name	Field						Comments
(Field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Table 1.1 RISC-V instruction formats.

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`** , and **`or`**.

A typical instance of such an instruction is:

```
add x1,x2,x3
```

, which reads **`x2`** and **`x3`** and writes the sum into **`x1`** .

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.

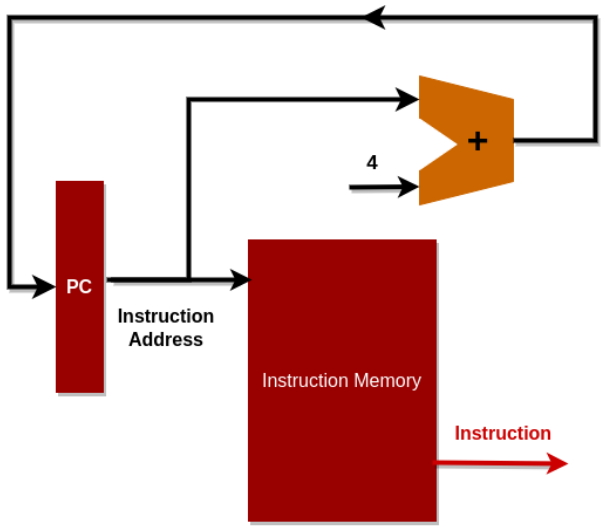


Figure 1.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

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.

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. For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers.

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.

The register file **always outputs** the contents of whatever register numbers are on the **Read** register inputs. Writes, however, are controlled by the **write control signal**, which must be **asserted** for a write to occur at the clock edge.

Figure 1.7 shows the result; we need a total of three inputs (two for register numbers and one for data) and two outputs (both for data). The register number inputs are **5 bits wide** to specify one of **32 registers** ($32 = 2^5$), whereas the **data input and two data output** buses are each **64 bits wide**.

Figure 1.7 shows the ALU, which takes two 64-bit inputs and produces a 64-bit result, as well as a 1-bit signal if the result is **`0`**.

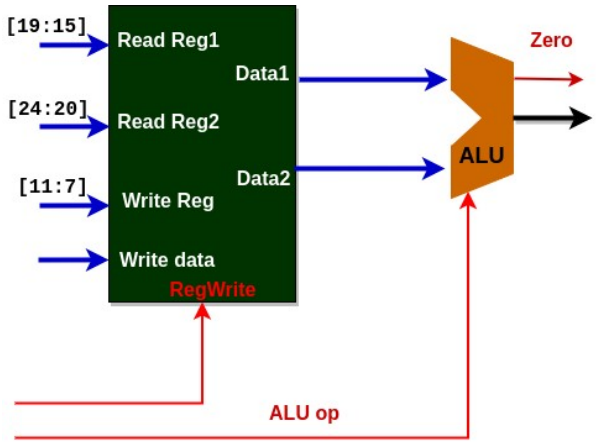


Figure 1.7 The two elements needed to implement **R-format ALU** operations are the register file and the ALU.

The **register file** contains all the registers and has **two read ports and one write port**.
The **register file always outputs the contents** of the registers corresponding to the **`Read`** register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that **writes are edge-triggered**, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) **must be valid at the clock edge**.

Since writes to the **register file are edge-triggered**, our design **can legally read and write the same register within a clock cycle**: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle.
The inputs carrying the **register number** to the register file **are all 5 bits wide**, whereas the lines carrying data values are 64 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide.
We will use the **`Zero`** **detection output** of the ALU shortly **to implement conditional branches**.

Next, consider the RISC-V **load** register and **store** register instructions, which have the general form:

`ld x1,offset(x2)` or **`sd x1,offset(x2)`** .

These instructions **compute** a **memory address by adding the base register**, which is **`x2`** , to the **12-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 **`x1`** . If the instruction is a **load**, the value read from memory must be **written** into the register file in the specified register, which is **`x1`** . Thus, we will need both the register file and the ALU from **Figure 1.7**.

In addition, we will need a unit to **sign-extend the 12-bit offset field** in the instruction **to a 64-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. **Figure 1.8** shows these two elements.

The **`beq`** instruction has three operands, two registers that are compared for equality, and a **12-bit offset** used to compute the **branch target address** relative to the branch instruction address.
Its form is:

`beq x1,x2,offset` .

To implement this instruction, we must compute the **branch target address** by **adding** the **sign-extended offset** field of the instruction **to the PC**.

There are **two details** in the definition of branch instructions to which we must pay attention:

1. The instruction set architecture specifies that the **base for the branch address** calculation is the address of the branch instruction.
2. The architecture also states that the **offset field is shifted left 1** bit so that it is a **half word offset**; this shift **increases** the effective **range of the offset** field **by a factor of 2**.

To deal with the latter complication, we will need to **shift the offset field by 1**. As well as computing the **branch target address**, we must also determine whether the next instruction is the **instruction that follows sequentially** or the instruction at the **branch target address**.
When the condition is true (i.e., two operands are equal), the **branch target address** becomes the **new `PC`**, and we say that the **branch is taken**.
If the operand **is not zero**, 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**.

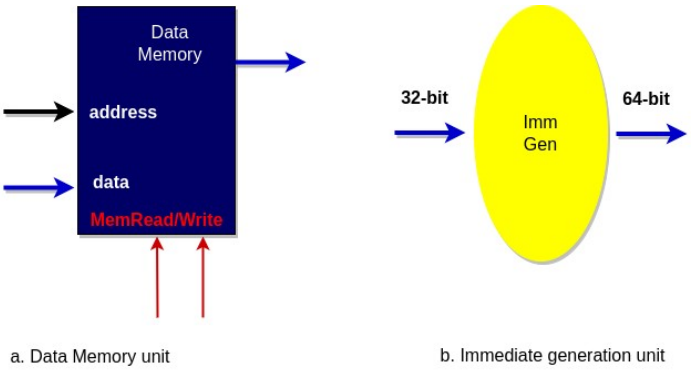


Figure 1.8 Data Memory and Immediate generation unit.

> The two units needed to implement loads and stores in addition to the register file and ALU of **Figure 1.8**, are the data memory unit and the **immediate generation unit**. The memory unit is a **state element** with inputs for the address and the write data, and a **single output for the read result**. There are separate read and write controls, although **only one of these may be asserted on any given clock**. The memory unit **needs a read signal**, since, unlike the register file, reading the value of an **invalid address** can cause problems. The immediate generation unit (**`ImmGen`**) has a **32-bit instruction as input** that selects a **12-bit field** for **load**, **store**, and **branch if equal** that is **sign-extended into a 64-bit** result appearing on the **output**. We assume the **data memory is edge-triggered for writes**. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could **easily be adapted to work with real memory chips**.

Thus, the branch datapath must do two operations: **compute the branch target** address and **test the register contents**.

Figure 1.9 shows the structure of the datapath segment that handles branches. To compute the branch target address, the branch datapath includes an immediate generation unit, from **Figure 1.8** and an adder. To perform the compare, we need to use the register file shown in **Figure 1.7a** to supply two register operands (although we will not need to write into the register file).

In addition, the equality comparison can be done using the **ALU**. Since that **ALU** provides an output signal that indicates whether the result was **0**, we can send both register operands to the **ALU** with the control set to subtract two values.
If the **`Zero`** signal out of the **ALU** unit is asserted, we know that the register values are equal. Although the Zero output always signals if the result is 0, we will be using it only to implement the equality test of conditional branches. Later, we will show exactly how to connect the control signals of the **ALU** for use in the datapath.
The **branch instruction operates by adding the PC with the 12 bits of the instruction shifted left by 1 bit**. Simply concatenating **0** to the branch offset accomplishes this shift.

1.3.1 Creating a Single Datapath

Now that we have examined the datapath components needed for the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation.

This **simplest datapath will attempt to execute all instructions in one clock cycle**. This design means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated. We therefore need a memory for **instructions separate** from one for **data**. Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows.

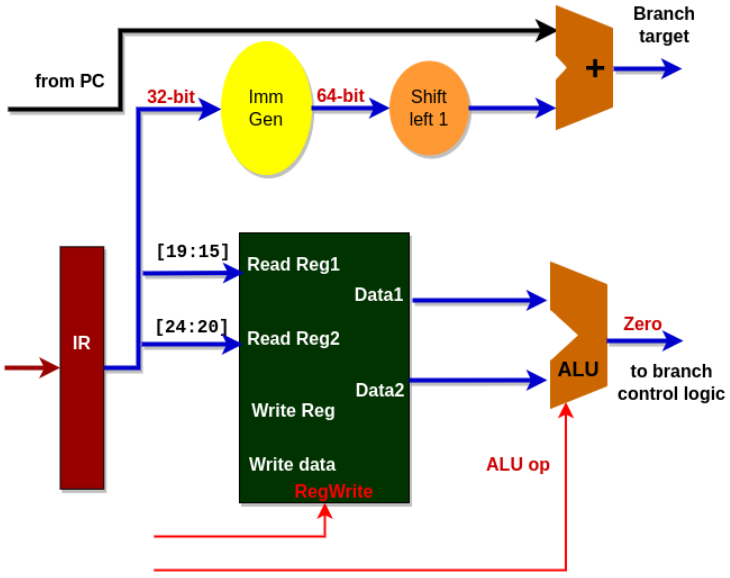


Figure 1.9 The **datapath for a branch** uses the ALU to evaluate the branch condition and a **separate adder** to compute the branch target as the sum of the PC and the **sign-extended 12 bits** of the instruction (the branch displacement), shifted left 1 bit.

> The unit labeled **Shift left 1** is simply a routing of the signals between input and output that a**dds 0 to the low-order end** of the sign-extended offset field; **no actual shift hardware is needed**, since the **amount of the "shift" is constant**. Since we know that the offset was **sign-extended from 12 bits**, the shift will **throw away only "sign bits."** Control logic is used to decide whether the incremented **`PC`** or branch target should replace the **`PC`**, based on the **`Zero`** output of the ALU.

To share a datapath element between two different instruction classes, we may need to allow multiple connections to the input of an element, using a multiplexor and control signal to select among the multiple inputs.
Now we can combine all the pieces to make a simple datapath for the core RISC-V architecture by adding the datapath for instruction fetch (**Figure 1.6**), the datapath from **R-type** and memory instructions (**Figure 1.10**), and the datapath for branches (**Figure 1.9**). **Figure 1.11** shows the datapath we obtain by composing the separate pieces. The branch instruction uses the main ALU to compare two register operands for equality, so we must keep the adder from **Figure 1.9** for computing the branch target address. An additional multiplexor is required to select either the sequentially following instruction address (**`PC+4`**) or the **branch target address** to be written into the PC.

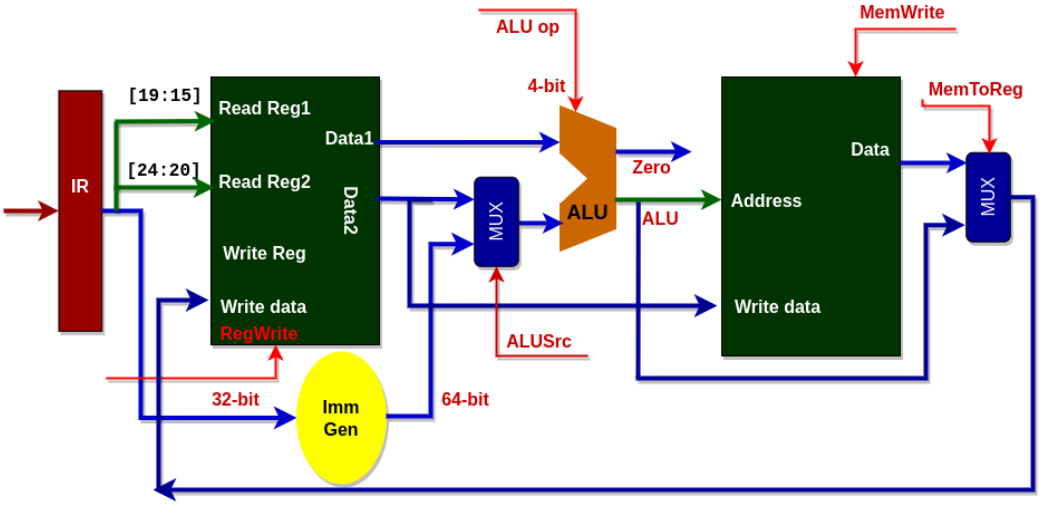


Figure 1.10 The datapath for the memory instructions and the **R-type** instructions. This example shows how a single datapath can be assembled from the pieces in **Figures 1.7** and **1.8** by **adding multiplexors**. Two multiplexors are needed, as described in the example.

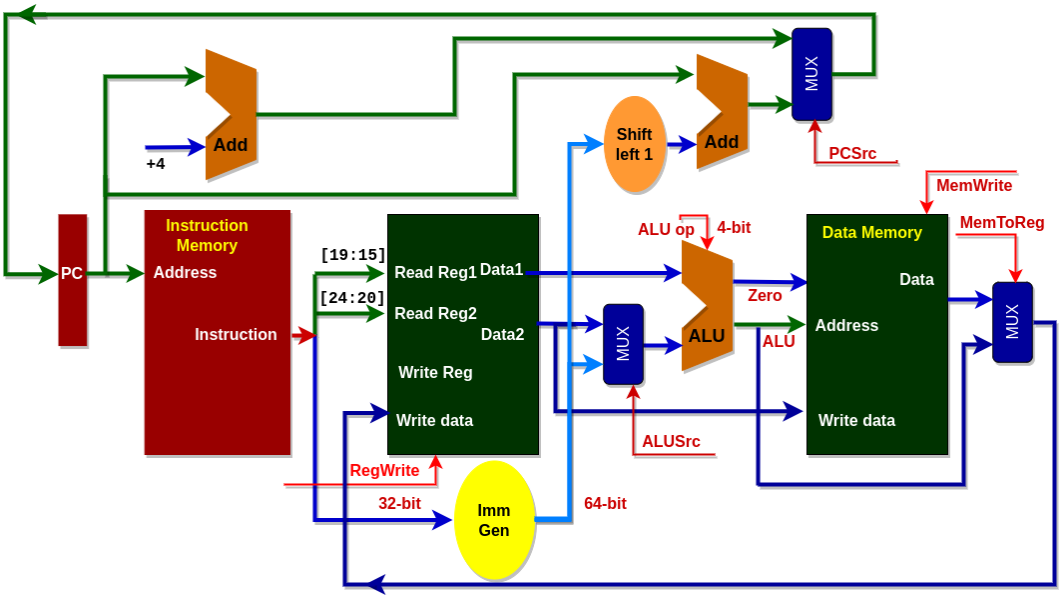


Figure 1.11 The simple datapath for the core RISC-V architecture combines the elements required by **different instruction classes-types: R-Type, IS-Type,SB-type**.

> The components come from **Figures 1.6, 1.9,** and **1.10**. This datapath can execute the basic instructions (**load-store register, ALU operations, and branches**) in a single clock cycle. Just one additional multiplexor is needed to integrate branches.

Now that we have completed this simple datapath, we can add the **control unit**. The control unit must be able to take inputs and **generate a write signal for each state element**, the **selector control for each multiplexor**, and the **ALU control**. The ALU control is different in a number of ways, and it will be useful to design it first before we design the rest of the control unit.

2. Control path for simple one cycle architecture

In this section, we build a **simple implementation of our RISC-V ISA subset**. The implementation is using the datapath of the last section and we add a **simple control function**.

This architecture covers **load doubleword** (`ld`), **store doubleword** (`sd`), **branch if equal** (`beq`), and the arithmetic-logical instructions **`add, sub, and,`** and **`or`** .

2.1 The ALU Control

ALU control lines	Operation
0000	and
0001	or
0010	add
0110	sub

Depending on the **instruction class-type**, the ALU will need to perform one of these four operations.

For **load** and **store** instructions, we use the ALU **to compute the memory address by addition**.

For the **R-type** instructions, the ALU needs to perform one of the four actions (**`and,or,add`**, or **`sub`**), depending on the value of the **7-bit `funct7` field** (**bits `31:25`**) and **3-bit `funct3` field** (**bits `14:12`**) in the instruction.
For the **conditional branch if equal** (**`beq`**) instruction, the ALU **subtracts two operands** and tests to see if the result is **0**.

We can generate the **4-bit ALU control input** using a small control unit that has as inputs the **`funct7`** and **`funct3`** fields of the instruction and a **2-bit control field**, which we call **`ALUOp`**.

`ALUOp` indicates whether the operation to be performed should be **add** (`00`) for **loads** and **stores**, **subtract and test if zero** (`01`) for **`beq`** , or be determined by the operation encoded in the **`funct7`** and **`funct3`** fields (`10`).
The output of the ALU control unit is a **4-bit signal** that directly controls the ALU by generating one of the 4-bit combinations shown previously.

In **Figure 2.1**, we show how to set the ALU control inputs based on the 2-bit **`ALUOp`** control, **`funct7`**, and **`funct3`** fields. Later in this section, we will see how the **`ALUOp`** bits are generated from the **main control unit**.

This style of using **multiple levels of decoding -** that is, the **main control unit generates** the **`ALUOp`** bits, which then are used as input to the ALU control that generates the **actual signals to control the ALU unit** - is a common implementation technique.
Using multiple levels of control can reduce the size of the main control unit. Using several smaller control units may also potentially **reduce the latency** of the control unit. Such optimizations are important, since the **latency of the control** unit is often a **critical factor** in determining the **clock cycle time**.

There are several different ways to implement the mapping from the **2-bit `ALUOp`** field and the **`funct`** fields to the **four ALU operation control bits**. Because only a small number of the possible **`funct`** field values are of interest and **`funct` fields are used only when the `ALUOp` bits equal `10`**, we can use a small piece of logic that recognizes the subset of possible values and generates the appropriate ALU control signals

Op code	ALUop	operation	funct7	funct3	ALU	ALU control
ld	00	load word	xxxxxxx	xxx	add	0010
sd	00	store word	xxxxxxx	xxx	add	0010
beq	01	branch =	xxxxxxx	xxx	sub	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	sub	0110
R-type	10	and	0000000	111	and	0000
R-type	10	or	0000000	110	or	0001

Figure 2.1 How the ALU control bits are set depends on the **`ALUOp`** control bits and the different opcodes for the **R-type** instruction.

The instruction, listed in the first column, determines the setting of the **ALUOp** bits. All the encodings are shown in binary. Notice that when the **ALUOp** code is **00** or **01**, the desired ALU action does not depend on the **funct7** or **funct3** fields; in this case, we say that we "don't care" about the value of the **opcode**, and the bits are shown as **Xs**. When the **ALUOp** value is **10**, then the **funct7** and **funct3** fields are used to set the ALU control input.

As a step in designing this logic, it is useful to create a **truth table** for the interesting combinations of **funct** fields and the **ALUOp** signals, as we've done in **Figure 2.2**; this truth table shows how the **4-bit ALU control** is set depending on these input fields.
Since the full **truth table is very large**, and we don't care about the value of the ALU control for many of these input combinations, we show only the truth table entries for which the ALU control must have a specific value.

We will use this practice of showing only the **truth table entries** for outputs that **must be asserted** and not showing those that are all deasserted or **don't** care.

Because in many instances we do not care about the values of some of the inputs, and because we wish to keep the tables compact, we also include don't-care terms.

A don't-care term in this truth table (represented by an **X** in an input column) indicates that t**he output does not depend on the value of the input** corresponding to that column.
For example, when the **ALUOp** bits are **00**, as in the first row of **Figure 2.2**, we always set the ALU control to **0010**, independent of the **funct** fields. In this case, then, the **funct** inputs will be don't cares in this line of the truth table.

2.2 Designing the main control unit

Now that we have described how to design an ALU that uses the opcode and a 2-bit signal as its control inputs, we can return to looking at the rest of the control.
To start this process, let's identify the fields of an instruction and the control lines that are needed for the datapath we constructed in **Figure 1.11**. To understand how to connect the fields of an instruction to the datapath, it is useful to review the formats of the four instruction classes: **arithmetic**, **load**, **store**, and **conditional** branch instructions. **Figure 2.3** shows these formats.

ALUOp		**Funct7 field**						**Funct3 field**				
ALUOp1	**ALUOp0**	**I[31]**	**I[30]**	**I[29]**	**I[28]**	**I[27]**	**I[26]**	**I[25]**	**I[14]**	**I[13]**	**I[12]**	**Operation**
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0010
1	X	0	0	0	0	0	0	0	1	1	1	0010
1	X	0	0	0	0	0	0	0	1	1	0	0010

Figure 2.2 The truth table for the 4 ALU control bits (called **Operation**).

The inputs are the **ALUOp** and **funct** fields. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the **ALUOp** does not use the encoding **11**, so the truth table can contain entries **1X** and **X1**, rather than **10** and **01**. While **we show all 10 bits of funct fields**, note that the only bits with different values for the four **R-format instructions** are bits **30, 14, 13,** and **12**.
Thus, we only need these **4 funct field bits as input for ALU control instead of all 10**.

Name	**Field**						**Comments**
(Field size)	**7 bits**	**5 bits**	**5 bits**	**3 bits**	**5 bits**	**7 bits**	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Figure 2.3 The **four instruction classes** (**arithmetic**, **load**, **store**, and **conditional branch**) use **four different instruction formats**.

(a) Instruction format for **R-type arithmetic instructions** (`opcode= 51`$_{10}$), which have three register operands: `rs1, rs2,` and `rd`. Fields `rs1` and `rs2` are **sources**, and `rd` is the **destination**.
The ALU function is in the `funct3` and `funct7` fields and is decoded by the ALU control design in the previous section. The **R-type** instructions that we implement are `add, sub, and,` and `or`.

(b) Instruction format for **I-type load instructions** (`opcode=3`$_{10}$). The register `rs1` is the **base register** that is **added to the 12-bit immediate field** to form the **memory address**. Field `rd` is the **destination** register for the loaded value.

(c) Instruction format for **S-type store** instructions (`opcode = 35`$_{10}$). The register `rs1` is the **base register** that is added to the **12-bit immediate field** to form the **memory address**. (The **immediate field is split** into a **7-bit piece** and a **5-bit piece**.) Field `rs2` is the **source** register whose value should be stored into memory.

(d) Instruction format for **SB-type** conditional branch instructions (`opcode = 99`$_{10}$). The registers `rs1` and `rs2` are **compared**. The **12-bit immediate address field is sign-extended**, **shifted left 1 bit**, and **added to the PC** to compute the **branch target address**.

There are **several major observations** about this instruction format that we will rely on:

- The `opcode` **field** is **always** in bits `6:0`. Depending on the opcode, the `funct3` field (`bits 14:12`) and `funct7` field (`bits 31:25`) serve as an **extended opcode field**.
- The **first register operand** is **always** in bit positions `19:15` (`rs1`) for **R-type** instructions and branch instructions. This field **also specifies the base register for load and store** instructions.
- The **second register operand** is always in bit positions `24:20` (`rs2`) for **R-type** instructions and **branch instructions**. This field also **specifies the register operand** that gets **copied to memory** for **store** instructions.
- **Another operand** can also be a **12-bit offset** for **branch** or **load-store** instructions.
- The **destination register is always** in bit positions `11:7` (`rd`) for **R-type** instructions and **load** instructions.

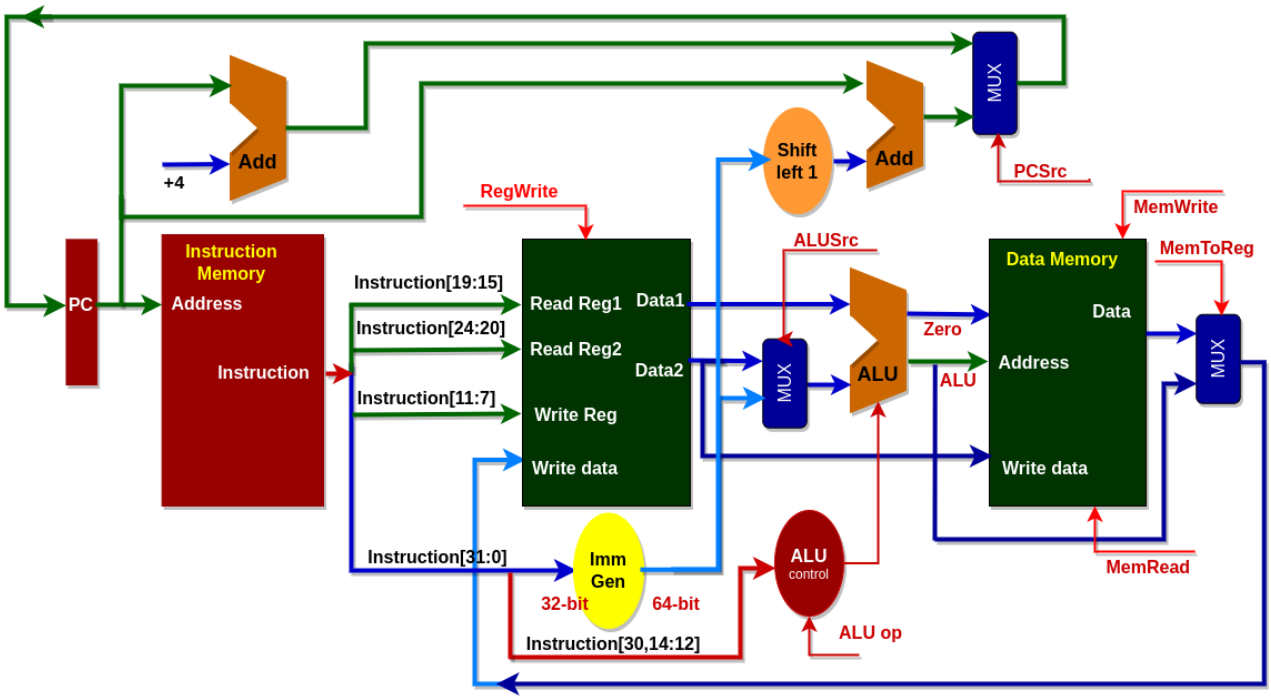


Figure 2.4 The datapath of **Figure 1.11** with all necessary multiplexors and **all control lines identified**.

> The control lines are shown in **color (red)**. The ALU control block has also been added, which depends on the `funct3` field and part of the `funct7` field. The **PC** does not require a write control, since it is written once at the end of every clock cycle; the **branch control logic** determines whether it is written with the **incremented PC** or the **branch target address**.

Using this information, we can add the instruction labels to the simple datapath.

Figure 2.4 shows these additions plus the ALU control block, the write signals for state elements, the read signal for the data memory, and the control signals for the multiplexors. Since all the multiplexors have two inputs, they each require a **single control line**.

Figure 2.4 shows **six single-bit control lines** plus the **2-bit `ALUOp`** control signal. We have already defined how the **`ALUOp`** control signal works, and it is useful to define what the six other control signals do informally before we determine how to set these control signals during instruction execution.

Figure 2.5 describes the function of these **six control lines**.

Now that we have looked at the function of each of the control signals, we can look at how to set them.
The control unit can set all but one of the control signals based solely on the opcode and **funct** fields of the instruction. The **`PCSrc`** control line is the exception. That control line should be asserted if the instruction is **branch if equal** (a decision that the control unit can make) and the **`Zero` output** of the ALU, which is used for the **equality test,** is asserted. To generate the **`PCSrc`** signal, we will need to AND together a signal from the control unit, which we call **`Branch`**, with the **`Zero`** signal out of the ALU.

Signal name	Effect when deasserted	Effect when asserted
`RegWrite`	None.	The register on the **Write register input** is written with the value on the **Write data input**
`ALUSrc`	The second ALU operand comes from the second register file output (**`Data 2`**)	The second ALU operand is **sign-extended**, 12-bits of the instruction
`PCSrc`	The PC is replaced by the output of the adder that computes **PC+4**	The **PC** is replaced by the output of the adder that computes the **branch target**
`MemRead`	None.	Data memory content designated by the **Address input** are put on the **Read data output**.
`MemWrite`	None.	Data memory content designated by the **Address input** are replaced by the value on the **Write data input**.
`MemToReg`	The value fed to the register Write **data comes from the ALU**.	The value fed to the register **Write data input comes from the data memory**.

Figure 2.5 The effect of each of the six control signals.

> When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems.

These **eight control signals** (six from Figure 2.5 and two for **`ALUOp`**) can now be set based on the input signals to the control unit, which are the opcode bits **`6:0`**.

Figure 2.6 shows the datapath with the control unit and the control signals.

Before we try to write a set of equations or a truth table for the control unit, it will be useful to try to **define the control function informally**. Because the setting of the control lines depends only on the **`opcode`**, we define whether each control signal should be **`0`**, **`1`**, or don't care (**`X`**) for each of the opcode values.

Figure- Table 2.7 defines how the control signals should be set for each opcode.

2.3 Operation of the Datapath

With the information contained in **Figures 2.5** and **2.7**, we can **design the control unit logic**, but before we do that, let's look at how each instruction uses the datapath. In the next few figures, we show the flow of three different instruction classes through the datapath.

The asserted control signals and active datapath elements are highlighted in each of these. Note that a multiplexor whose control is `0` has a definite action, even if its control line is not highlighted. Multiple-bit control signals are highlighted if any constituent signal is asserted.

Figure 2.8 shows the operation of the datapath for an **R-type instruction**, such as:

```
add x1, x2, x3
```

Although everything occurs **in one clock cycle**, we can think of **four steps** to execute the instruction; these steps are ordered by the **flow of information**:

1. The instruction **is fetched**, and the `PC` is incremented.
2. Two registers, `x2` and `x3` , **are read** from the register file; also, the main control unit computes the setting of the control lines during this step.
3. The ALU operates on the data read from the register file, using portions of the opcode to generate the ALU function.
4. The result from the ALU **is written** into the destination register (`x1`) in the register file.

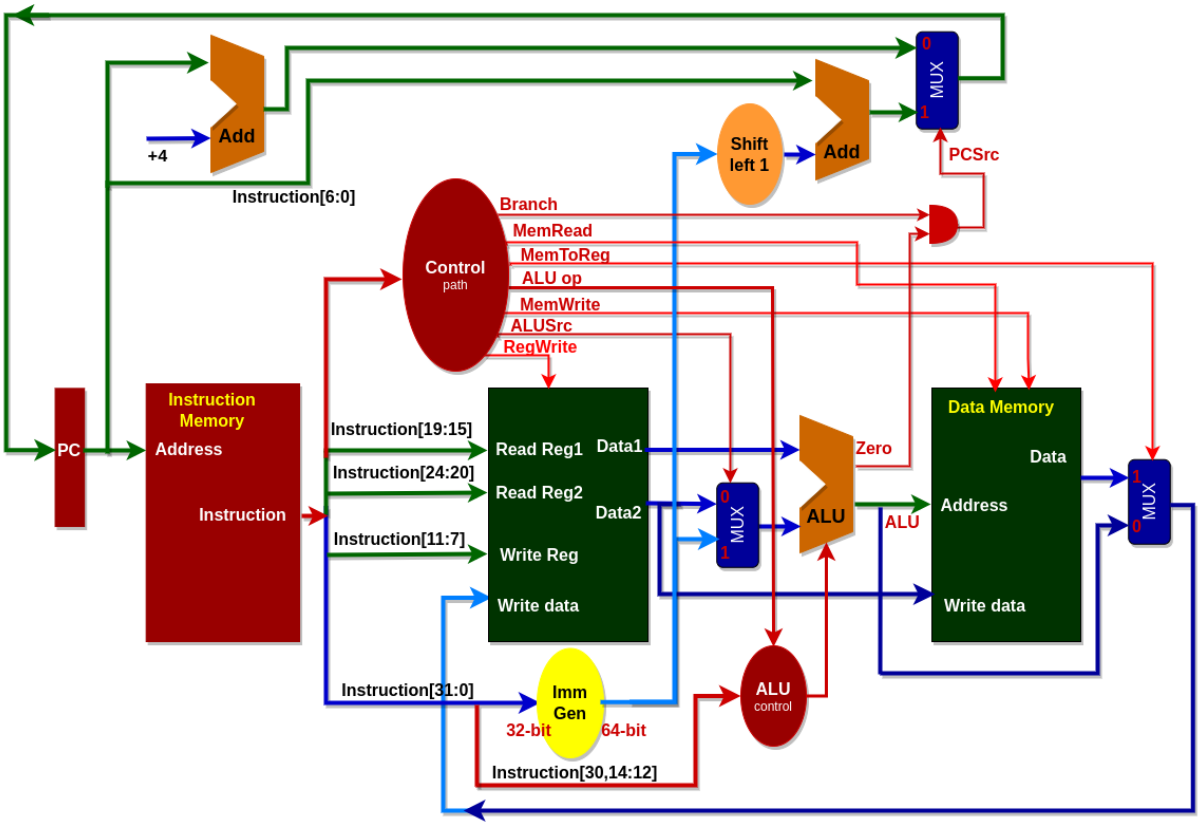


Figure 2.6 The simple datapath **with the control unit**.

The **input** to the control unit is the **7-bit opcode field** from the instruction.
The outputs of the control unit consist of **two 1-bit signals** that are used to control **multiplexors** (`ALUSrc` and `MemtoReg`), **three signals** for controlling **reads and writes in the register file and data memory** (`RegWrite`, `MemRead`, and `MemWrite`), a **1-bit signal** used in determining whether to possibly branch (`Branch`), and a **2-bit control signal** for the ALU (`ALUOp`). An **AND gate** is used to combine the branch control signal and the **Zero** output from the ALU; the **AND** gate output controls the **selection of the next PC**. Notice that `PCSrc` is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures.

Instruction	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Figure 2.7 The setting of the control lines is completely determined by the opcode fields of the instruction.

The first row of the table corresponds to the **R-format instructions** (**`add`**, **`sub`**, **`and`**, and **`or`**). For all these instructions, the source register fields are **`rs1`** and **`rs2`**, and the destination register field is **`rd`**; this defines how the signals **`ALUSrc`** is set.
Furthermore, an **R-type instruction** writes a register (**`RegWrite=1`**), but **neither reads nor writes data memory**.
When the **`Branch`** control signal is **`0`**, the **`PC` is unconditionally replaced with `PC+4`**; otherwise, the **`PC`** is replaced by the **branch target** if the **`Zero`** output of the ALU i**s also high**.

The **`ALUOp`** field for **R-type instructions** is set to **`10`** to indicate that the ALU **control** should be generated **from the `funct` fields**.

The second and third rows of this table give the control signal settings for **`ld`** and **`sd`**. These **`ALUSrc`** and **`ALUOp`** fields are set to perform the address calculation. The **`MemRead`** and **`MemWrite`** are set to perform the **memory access**.

Finally, **`RegWrite`** is set for a **load** to cause the result to be **stored** in the **`rd`** register. The **`ALUOp`** field for branch is **set for subtract** (ALU control **`= 01`**), which is used **to test for equality**.

Notice that the **`MemtoReg`** field **is irrelevant** when the **`RegWrite`** signal is **`0`**: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry **`MemtoReg`** in the last two rows of the table is **replaced with `X`** for don't care. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

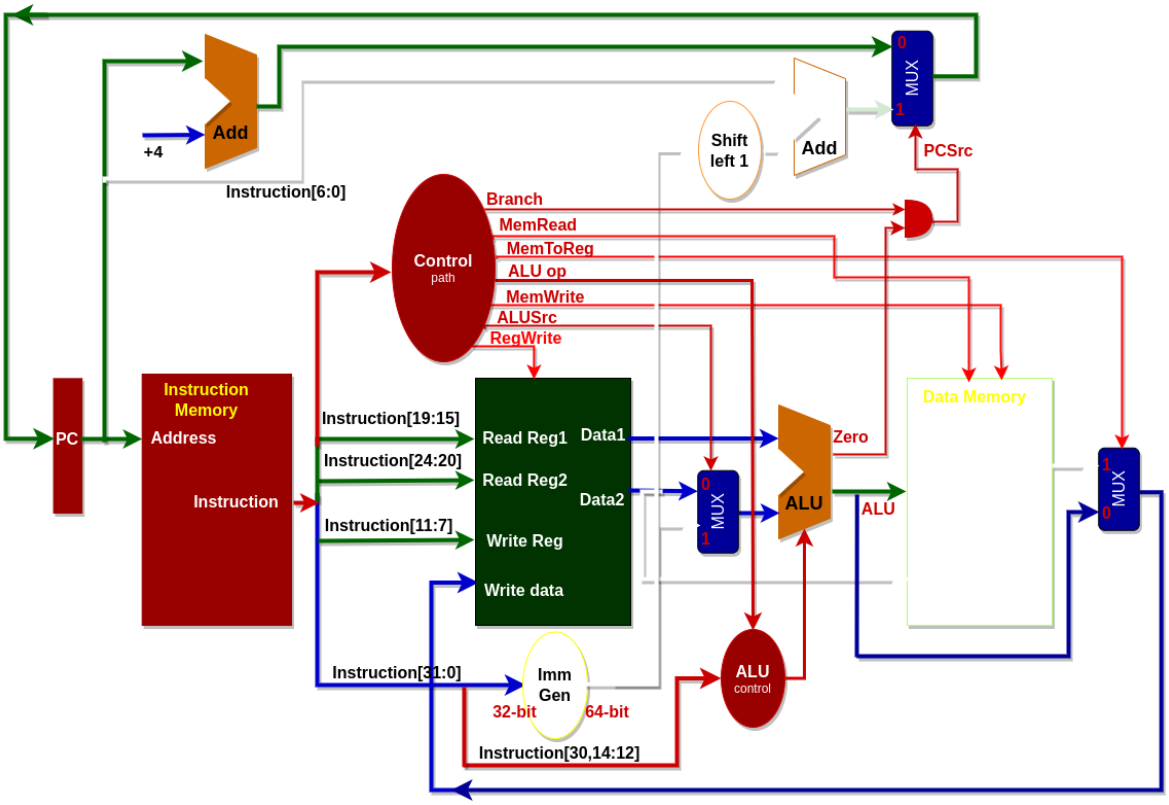


Figure 2.8 The **datapath operation** for an **R-type** instruction, such as **`add x1,x2,x3`**. The control lines, datapath units, and connections that are active **are highlighted**.

Similarly, we can illustrate the execution of a load register, such as:

```
ld x1,offset(x2)
```

in a style similar to **Figure 2.8**. **Figure 2.9** shows the active functional units and asserted control lines for a load.

We can think of a **load instruction** as operating in **five steps** (similar to how the **R-type executed in four**):

1. An instruction is fetched from the instruction memory, and the `PC` is incremented.
2. A register (`x2`) value **is read** from the register file.
3. The **ALU computes** the **sum of the value read from the register file and the sign-extended 12 bits** of the instruction (**offset**).
4. The **sum** from the ALU is **used as the address** for the **data memory**.
5. The **data from the memory** unit is written **into the register file** (`x1`).

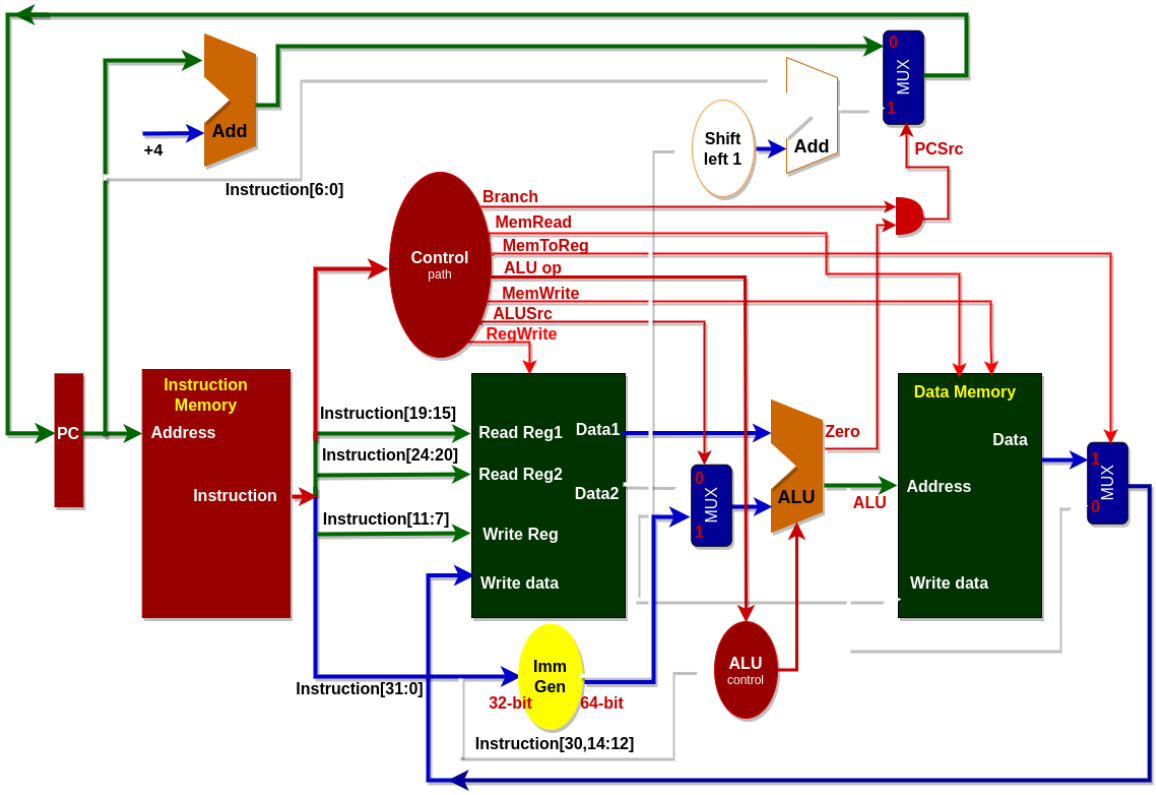


Figure 2.9 The **datapath in operation** for a **load** instruction.

> The control lines, datapath units, and connections that are active are highlighted. A **store instruction** would operate very similarly. The main difference would be that the memory control would indicate a **write** rather than a read, the **second register value read would be used for the data to store**, and the operation of **writing** the data memory value to the register file **would not occur**.

Finally, we can show the operation of the **branch-if-equal** instruction, such as:

```
beq x1,x2,offset
```

It operates **much like an R-format** instruction, but the ALU output is used to determine whether the PC is written with `PC+4` or the branch target address. Figure 2.10 shows the four steps in execution:

1. An instruction is **fetched** from the instruction memory, and the `PC` **is incremented**.
2. Two registers, `x1` and `x2` , are **read** from the register file.
3. The ALU **subtracts** one data value from the other data value, both read from the register file. The value of `PC` is added to the sign-extended, 12 bits of the instruction (`offset`) l**eft shifted by one**; the result is the **branch target address**.
4. The `Zero` **status** information from the ALU is used to decide which adder result to store in the `PC`.

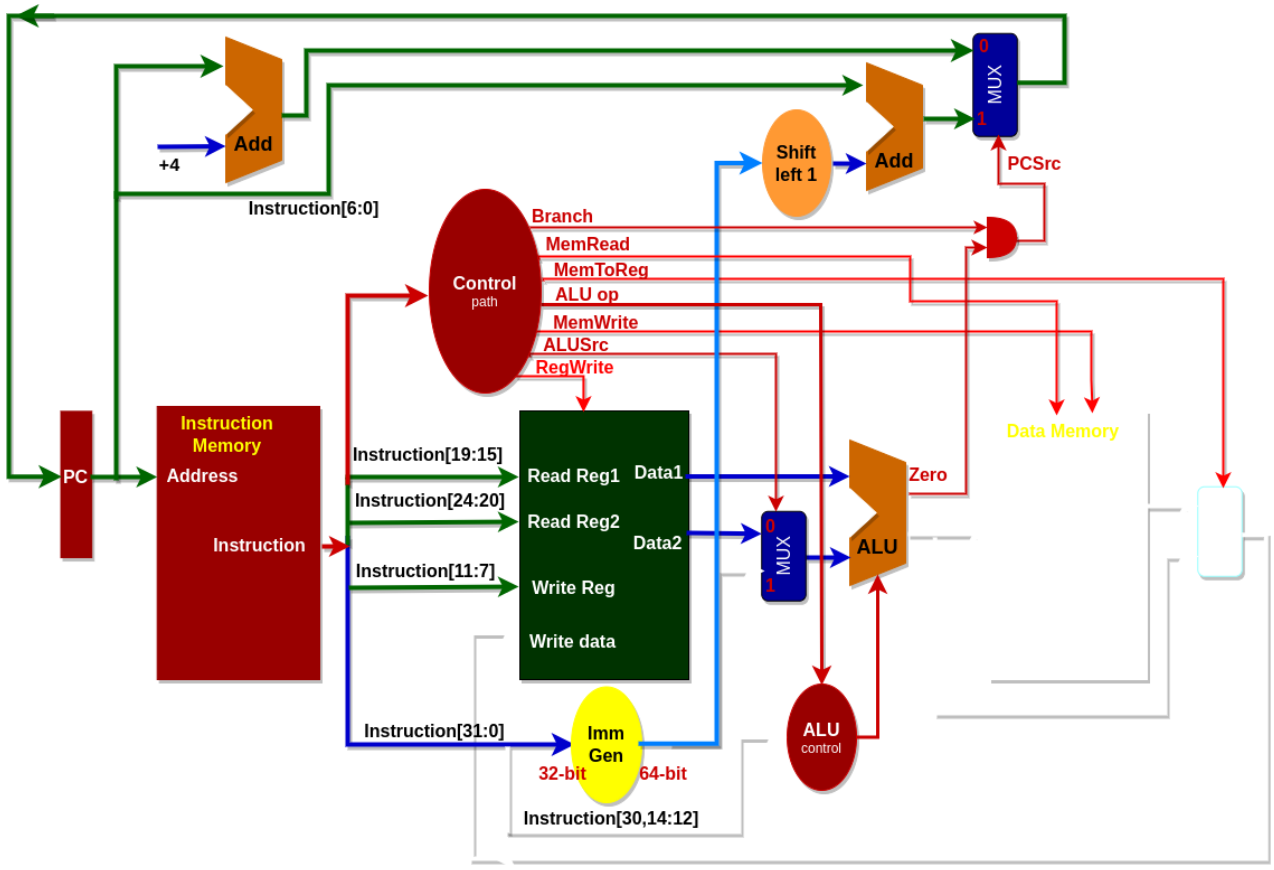


Figure 2.10 The **datapath operation** for a **branch-if-equal** – `beq` instruction.

The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the `Zero` output is used to select the **next program counter from between the two candidates**.

2.4 Finalizing Control

Now that we have seen how the instructions operate in steps, let's continue with the control implementation. The control function can be precisely defined using the contents of **Figure 2.7**. The outputs are the control lines, and the inputs are the opcode bits. Thus, we can create a truth table for each of the outputs based on the binary encoding of the opcodes.
Figure 2.11 defines the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs. It completely specifies the control function, and we can implement it directly in gates in an automated fashion.

2.4.1 Why a Single-Cycle Implementation is not used today

Although the single-cycle design **will work correctly**, it **is too inefficient** to be used in modern designs. To see why this is so, notice that the **clock cycle must have the same length for every instruction** in this single-cycle design. Of course, **the longest possible path in the processor determines the clock cycle**.

This path is most likely a **load** instruction, which uses **five functional units in series**:

1. the instruction memory,
2. the register file,
3. the ALU,
4. the data memory, and
5. the register file.

Although the **CPI** (**Clock cycles Per Instruction**) is **1** , the overall performance of a single-cycle implementation is likely to be poor, since the **clock cycle is too long**.

The penalty for using the single-cycle design with a fixed clock cycle is significant, but might be considered acceptable for this small instruction set. Historically, early computers with very simple instruction sets did use this implementation technique.
However, if we tried to implement the floating-point unit or an instruction set with more complex instructions, this single-cycle design **wouldn't work well at all**.
Because we must assume that the **clock cycle is equal to the worst-case delay for all instructions**, it's useless to try implementation techniques that reduce the delay of the common case but do not improve the worst-case cycle time.

A single- cycle implementation thus violates the great idea of making the common case fast.

In next section, we'll look at another implementation technique, called **pipelining**, that uses a datapath very similar to the single-cycle datapath but is much more efficient by having a much higher throughput.

Pipelining improves efficiency by **elaborting multiple instructions simultaneously**.

2.5 An overview of Pipelining

Pipelining is an implementation technique in which multiple instructions **are overlapped in execution**.

This section relies heavily on one analogy to give an overview of the pipelining terms and issues.

RISC-V RV64I instructions classically take five basic steps (stages):

1. **Fetch instruction** from memory.
2. **Read registers** and **decode** the instruction.
3. **Execute the operation** or **calculate an address**.
4. **Access an operand** in data **memory** (**if necessary**).
5. **Write the result** into a register (**if necessary**).

The number of additional steps (stages) depends on the extensions.

2.5.1 Example S7 core - RV64GC

The following is the pipeline of well known S7 RISC-V core from SiFive. Four S7 cores are integrated in **JH7110** SoC. JH7110 is implemented in **StarFive Visionfive 2** board.

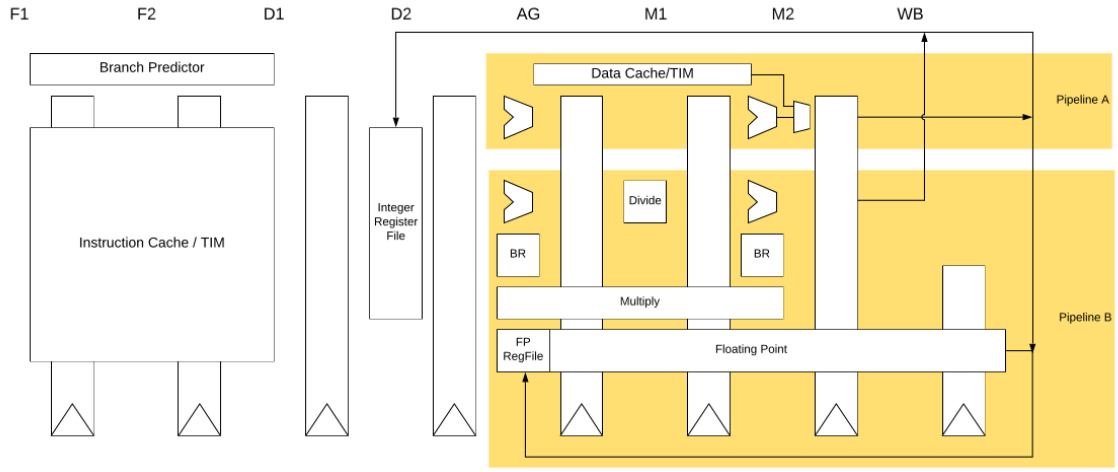


Figure 2.11 S7 (JH7110 – SiFive SoC) - with 8 pipeline stages

The **S7** execution unit is a **dual-issue, in-order pipeline**.
The pipeline comprises **eight stages**:

- two stages of instruction fetch (**F1** and **F2**),
- two stages of instruction decode (**D1** and **D2**), address generation (**AG**),
- two stages of data memory access (**M1** and **M2**), and register write-back (**WB**).

The pipeline has a peak execution rate of two instructions per clock cycle, and **is fully bypassed** so that **most instructions have a one-cycle result latency**:

• **Integer arithmetic and branch** instructions can execute in either the **AG** or **M2** pipeline stage. If such an instruction's operands are available when the instruction enters the AG stage, then it executes in AG; otherwise, it executes in M2.
• **Loads** produce their result in the **M2** stage. There is no load-use delay for most integer instructions. However, effective addresses for memory accesses are always computed in the AG stage. Hence, loads, stores, and indirect jumps require their address operands to be ready when the instruction enters AG. If an address-generation operation depends upon a load from memory, then the load-use delay is two cycles.
• **Integer multiplication** instructions consume their operands in the AG stage and produce their results in the M2 stage. The **integer multiplier is fully pipelined**.
• **Integer division** instructions consume their operands in the AG stage. These instructions have **between a six-cycle and 68-cycle result latency**, depending on the operand values.

The pipeline only interlocks on read-after-write and write-after-write hazards, so instructions may be scheduled to avoid stalls.

The pipeline implements a flexible dual-instruction-issue scheme. Provided there are **no data hazards** between a pair of instructions, the two instructions may issue in the same cycle, provided the following constraints are met:

• At most one instruction accesses data memory.
• At most one instruction is a branch or jump.
• At most one instruction is an integer multiplication or division operation.
• Neither instruction explicitly accesses a CS

The RISC-V **pipeline we explore in this chapter has five stages**. The following example shows that pipelining speeds up instruction execution:

2.5.2 Single-Cycle versus Pipelined Performance

To make this presentation more concrete, let's create a pipeline. In this example, and in the rest of this section, we limit our attention to **seven instructions**:

- load doubleword : **`ld`** ,
- store doubleword: **`sd`** ,
- add: **`add`** ,
- subtract: **`sub`** ,
- AND: **`and`** ,
- OR: **`or`** , and
- branch if equa: **`beq`** .

Let us contrast the **average time between instructions** of a **single-cycle implementation**, in which all instructions take one clock cycle, to a **pipelined implementation**.

Let us assume that the operation times for the major functional units in this example are **200ps** for memory access for instructions or data, **200ps** for ALU operation, and **100ps** for **register file read or write**.
In the single-cycle model, every instruction takes exactly one **clock cycle**, so **the clock cycle must be stretched** to accommodate the **slowest instruction**.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
`lld`	200	100	200	200	100	**800ps**
`sd`	200	100	200	200		**700ps**
`add,sub,and,or`	200	100	200		100	**600ps**
`beq`	200	100	200			**500ps**

Figure 2.12 Total time for each instruction calculated from the time for each component.

This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

Figure 2.12 shows the time required for each of the seven instructions. The single-cycle design must allow for the slowest instruction—in **Figure 2.9** it is **`ld`** —so the time required for every instruction is **800ps**.

Figure 2.13 compares nonpipelined and pipelined execution of **three load register instructions**. Thus, the time between the first and fourth instructions in the nonpipelined design is **3x800ps** or **2400ps**.

All the pipeline stages take a single clock cycle, so the clock cycle must be **long enough to accommodate the slowest operation**. Just as the single-cycle design must take the **worst-case clock cycle of 800ps**, even though some instructions can be as fast as 500ps, the **pipelined execution clock cycle must have the worst-case clock cycle of 200ps**, even though some stages take only **100ps**.
In this case, the pipelining still offers a **fourfold performance improvement**: the time between the first and fourth instructions is **3x200ps** or **600ps**.

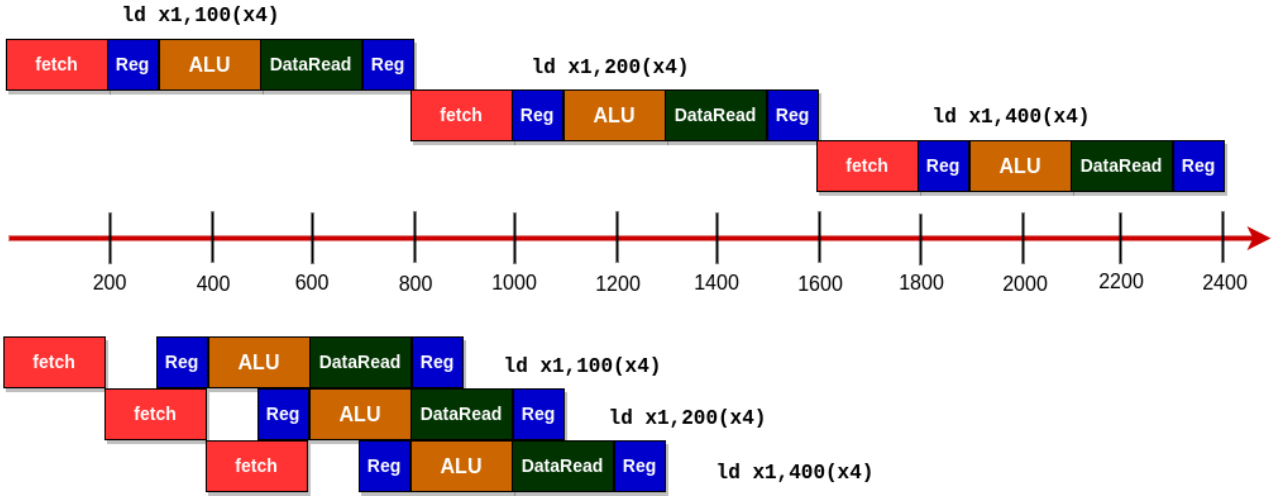


Figure 2.13 Single-cycle, nonpipelined execution (top) versus pipelined execution

Both architectures use the same hardware components, whose **time is listed in Figure 2.11**. In this case, we see a fourfold speed-up on average time between instructions, from **800ps** down to **200ps**. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access (**200ps**). We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half.

2.6 RISC-V ISA and pipelining

Even with this simple explanation of pipelining, we can get insight into the design of the RISC-V instruction set, which was designed for pipelined execution.

1. First, all RISC-V **instructions are the same length**. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage.
2. Second, RISC-V has just **a few instruction formats**, with the source and destination register fields being located in the same place in each instruction.
3. Third, **memory operands only** appear in **loads or stores** in RISC-V. This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage.

2.6.1 Pipeline Hazards

There are three types of pipeline hazards that provoke pipeline stall.

Structural Hazard appears when the hardware cannot support the combination of instructions that we want to execute in the same clock cycle (the same resources !). If the pipeline in **Figure 2.13** had a **fourth instruction**, we would see that in the same clock cycle, the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, one for the instructions and one for the data , our pipeline could have a structural hazard.

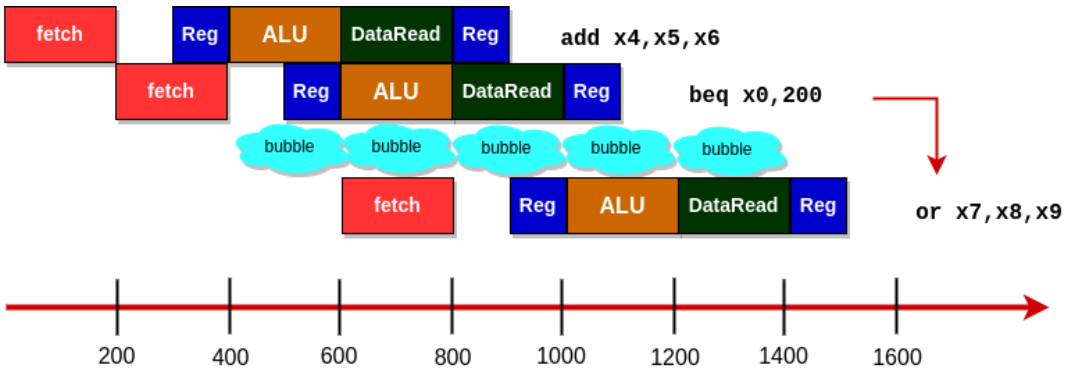
Data Hazards occur when the pipeline must be stalled because one step must wait for another to complete. Data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline .
For example, suppose we have an **add instruction followed immediately by a subtract instruction** that uses that sum.

```
add x19, x0, x1
sub x2, x19, x3
```

Without intervention, a data hazard could **severely stall the pipeline**. The add instruction doesn't write its result until the fifth stage, meaning that we would have to **waste three clock cycles** in the pipeline.
Solution to data hazerd: For the code sequence above, as soon as the ALU creates the sum for the **`add`**, we can supply it as an immediate input for the subtract (**`sub`**). Adding extra hardware (**result buffer**) to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.

Control Hazard is arising from the need to make a **decision based on the results of one instruction** while others are executing.

Control hazard may arise with the conditional branch instruction. Notice that we must begin fetching the instruction following the branch on the following clock cycle. Nevertheless, the pipeline cannot possibly know what the next instruction should be, since it only just received the branch instruction from memory.
One **possible solution is to stall immediately** after we fetch a branch (**`nop`**), waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.



2.14 Pipeline showing stalling on every conditional branch as solution to control hazards.

This example **assumes the conditional branch is taken**, and the instruction at the destination of the branch is the **`or`** instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated. The effect on performance, however, is the same as would occur if a bubble were inserted.

2.6.2 Pipeline Overview Summary

Pipelining is a **technique that exploits parallelism between the instructions** in a s**equential instruction stream**. Pipelining is **fundamentally invisible to the programmer**.

In the next section, we cover the concept of pipelining using the RISC-V instruction subset from the single-cycle implementation and show a simplified version of its pipeline.

2.6.2.1 Simple test:

For each code sequence below, state whether **it must stall**, can **avoid stalls using only forwarding**, or **can execute without stalling or forwarding**.

Sequence 1

```
ld      x10,0(x10)
add     x11,x10,x10
```

Sequence 2

```
add     x11,x10,x10
addi    x12,x10,5
addi    x14,x11,5
```

Sequence 3

```
addi    x11,x10,1
addi    x12,x10,2
addi    x13,x10,3
addi    x14,x10,4
addi    x15,x10,4
```

2.7 Implementation of pipelined datapath and control path

The following figure shows the single-cycle datapath from previous section with the pipeline stages identified. The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle.
Thus, we must separate the datapath into five pieces, with each piece named corresponding to a **stage** of instruction execution:

1. **`IF`:** **Instruction fetch**
2. `ID`: **Instruction decode** and register file read
3. `EX`: **Execution** or address calculation
4. `MEM`: **Data memory access**
5. `WB`: **Write back**

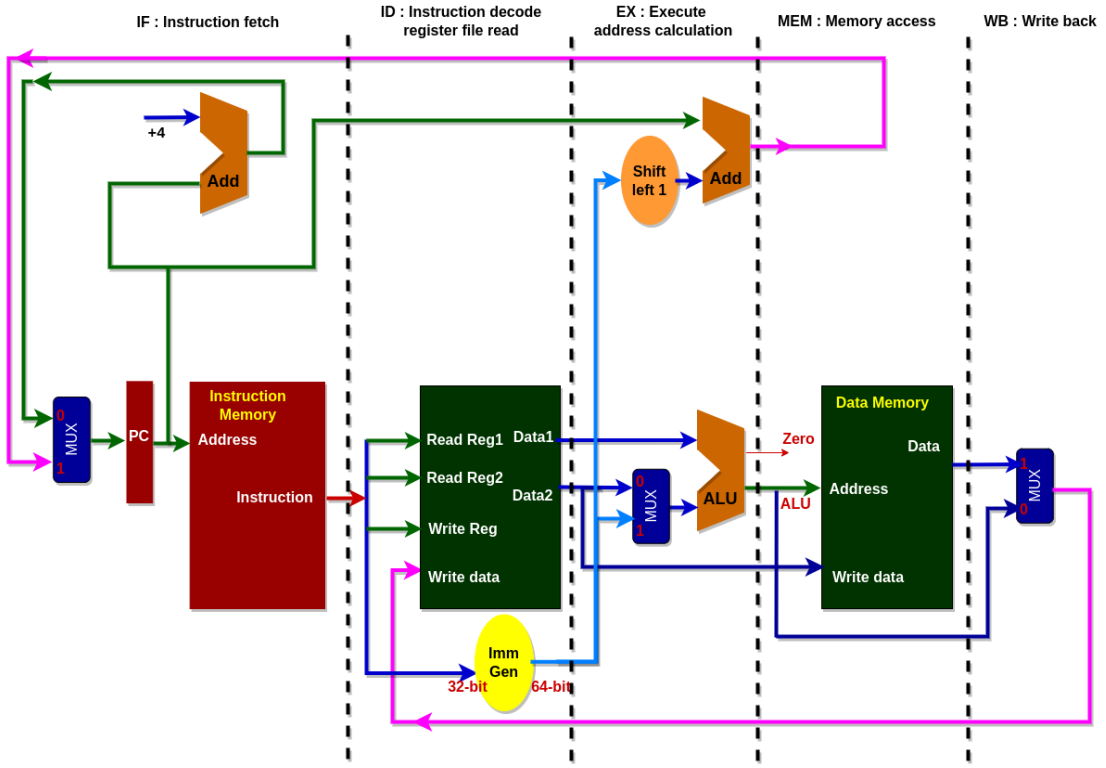


Figure 2.15 The single-cycle datapath with stages.

> Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left **to be written into the register file**. (Normally we use color lines for control, but these are data lines.)

In **Figure 2.15**, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the five stages as they complete execution.
There are, however, two exceptions to this left-to-right flow of instructions:

1. The **write-back stage**, which places the result back into the register file in the middle of the datapath (data hazard possible)
2. The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage (control hazard possible)

Data flowing from right to left do not affect the current instruction; these reverse data movements influence only later instructions in the pipeline. Note that the first right-to-left flow of data can lead to data hazards and the second leads to control hazards.

One way to show what happens in pipelined execution is to pretend that each instruction has its own datapath, and then to place these datapaths on a timeline to show their relationship.

To retain the value of an individual instruction for its other four stages, the value read from instruction memory **must be saved in a register**. Similar arguments apply to every pipeline stage, so **we must place registers wherever there are dividing lines between stages** in **Figure 2.16**.

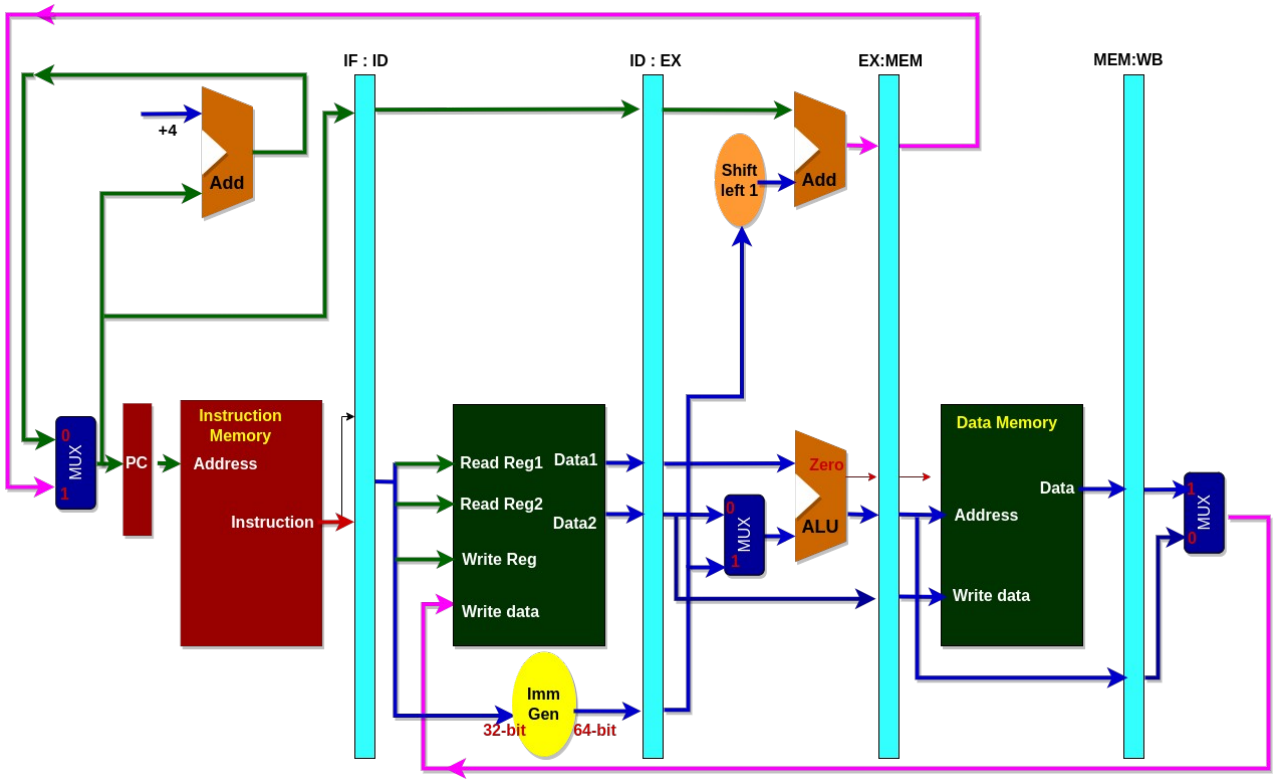


Figure 2.16 The pipelined version of the datapath in **Figure 2.15**

> The pipeline registers, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled **IF/ID** because it separates the instruction fetch and instruction decode stages. The registers must be **wide enough** to store all the data corresponding to the lines that go through them. For example, the **IF/ID register** must be **96 bits wide**, because it must hold both the **32-bit instruction** fetched from memory and the **incremented 64-bit `PC` address**. We will expand these registers over the course of this section, but for now the other three pipeline registers contain **256 (ID:EX)**, **193 (EX:MEM)** , and **128** bits (**MEM:WB**), respectively.

To show how the pipelining works we need to analize the pipeline stages;
The five stages are the following (depending on the instruction type):

1. **Instruction fetch** : The instruction is being read from memory using the address in the PC and then being placed in the **IF/ID pipeline register**. The `PC` address is **incremented by 4** and then written back into the `PC` to be ready for the next clock cycle. This **`PC` is also saved in the IF/ID pipeline register** in case it is needed later for an instruction, such as **`beq`** . The computer **cannot know** which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

2. **Instruction decode and register file read** : The bottom portion of Figure 2.15 shows the instruction portion of the **IF/ID pipeline register supplying the immediate field**, which is **sign-extended to 64 bits**, and the register numbers to read the two registers. All three values are stored in the **ID/EX pipeline register**, along with the **`PC`** address. We again transfer everything that might be needed by any instruction during a later clock cycle.

3. **Execute or address calculation**: The **load** instruction reads the contents of a **register** and the **sign-extended immediate from the ID/EX** pipeline register and adds them using the ALU. That sum is placed in the **EX/MEM pipeline register**.

4. **Memory access**: The **load** instruction is reading the data memory using the address from the EX/MEM pipeline register and loading the data into the **MEM/WB pipeline register**.

5. **Write-back**: The final step: **reading the data from the MEM/WB pipeline register** and writing it **into the register file**

2.7.1 Pipelined Control

Just as we added control to the single-cycle datapath we now add control to the pipelined datapath. We start with a simple design that views the problem **without the potential hazards !**
The first step is to label the control lines on the existing datapath. **Figure 2.17** shows those lines. As was the case for the single-cycle implementation, we assume that the PC is written on each clock cycle, so there is no separate write signal for the PC.
By the same argument, there are no separate write signals for the pipeline registers (I**F/ ID, ID/EX, EX/MEM**, and **MEM/WB**), since the pipeline registers are also written during each clock cycle.

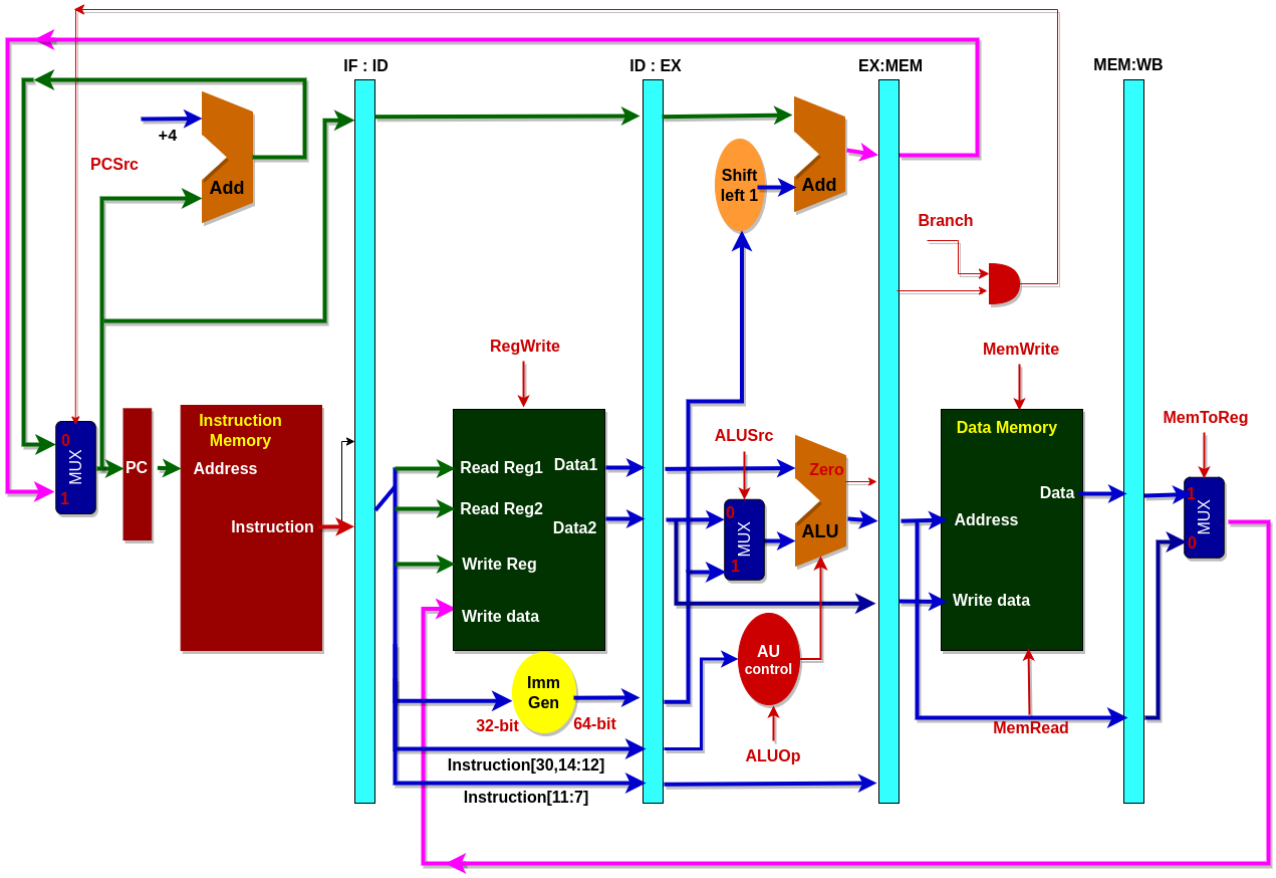


Figure 2.17 The pipelined datapath of **Figure 2.16** with the control signals identified.

> This datapath borrows the control logic for PC source, register destination number, and ALU control from previous section. Note that we now need **`funct`** fields of the instruction in the **EX** stage as input to ALU control, so these bits must also be included in the **ID/EX** pipeline register.

To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into **five groups** according to the pipeline stage.

1. **Instruction fetch**: The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.
2. **Instruction decode/register file read**: The two source registers are always in the same location in the RISC-V instruction formats, so there is nothing special to control in this pipeline stage.
3. **Execution/address calculation**: The signals to be set are **`ALUOp`** and **`ALUSrc`**. The signals select the ALU operation and either **`Read_data2`** or a **sign-extended immediate** as inputs to the ALU.
4. **Memory access**: The control lines set in this stage are **`Branch`**, **`MemRead`**, and **`MemWrite`**. The **branch if equal** (**`beq`**), **load** (**`ld`**), and **store** (**`sd`**) instructions set these signals, respectively. Recall that **`PCSrc`** selects the next sequential address unless control asserts **`Branch`** and the ALU result was **0**.
5. **Write-back**: The two control lines are **`MemtoReg`**, which decides between sending the ALU result or the memory value to the register file, and **`RegWrite`**, which writes the chosen value.

Since pipelining the datapath leaves the meaning of the control lines unchanged, we can use the same control values.

Op code	ALUop	operation	funct7	funct3	ALU	ALU control
ld	00	load word	xxxxxxx	xxx	add	0010
sd	00	store word	xxxxxxx	xxx	add	0010
beq	01	branch =	xxxxxxx	xxx	sub	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	sub	0110
R-type	10	and	0000000	111	and	0000
R-type	10	or	0000000	110	or	0001

Figure 2.18 (copy of Figure 2.1) How the ALU control bits are set depends on the **ALUOp** control bits and the different opcodes for the **R-type** instruction.

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the **Write register input** is written with the value on the **Write data input**
ALUSrc	The second ALU operand comes from the second register file output (Data 2)	The second ALU operand is **sign-extended**, 12-bits of the instruction
PCSrc	The PC is replaced by the output of the adder that computes **PC+4**	The **PC** is replaced by the output of the adder that computes the **branch target**
MemRead	None.	Data memory content designated by the **Address input** are put on the **Read data output**.
MemWrite	None.	Data memory content designated by the **Address input** are replaced by the value on the **Write data input**.
MemToReg	The value fed to the register Write **data comes from the ALU**.	The value fed to the register **Write data input comes from the data memory**.

Figure 2.19 (Copy of Figure 2.5) The effect of each of the six control signals.

The function of each of six control signals is defined. The ALU control lines (**ALUOp**) are defined in the second column . When a 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the **0** input.
Note that **PCSrc** is controlled by an **AND** gate .
If the **Branch** signal and the ALU **Zero** signal are both set, then **PCSrc** is 1; otherwise, it is **0**. Control sets the Branch signal only during a **beq** instruction; otherwise, **PCSrc** is set to **0**.

Instruction	Execution/Address			Memory Access		Write-Back	
	ALUOp	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemToReg
R-type	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Figure 2.20 The values of the control lines are the same as in Figure 2.6, but they have been shuffled into **three groups** corresponding to the **last three pipeline stages**.

Implementing control means setting the seven control lines to these values in each stage for each instruction. Since the rest of the control lines starts with the EX stage, we can create the control information during instruction decode for the later stages.

The simplest way to pass these control signals is to extend the pipeline registers to include control information. **Figure 2.21** shows the full datapath with the extended pipeline registers and with the control lines connected to the proper stage.

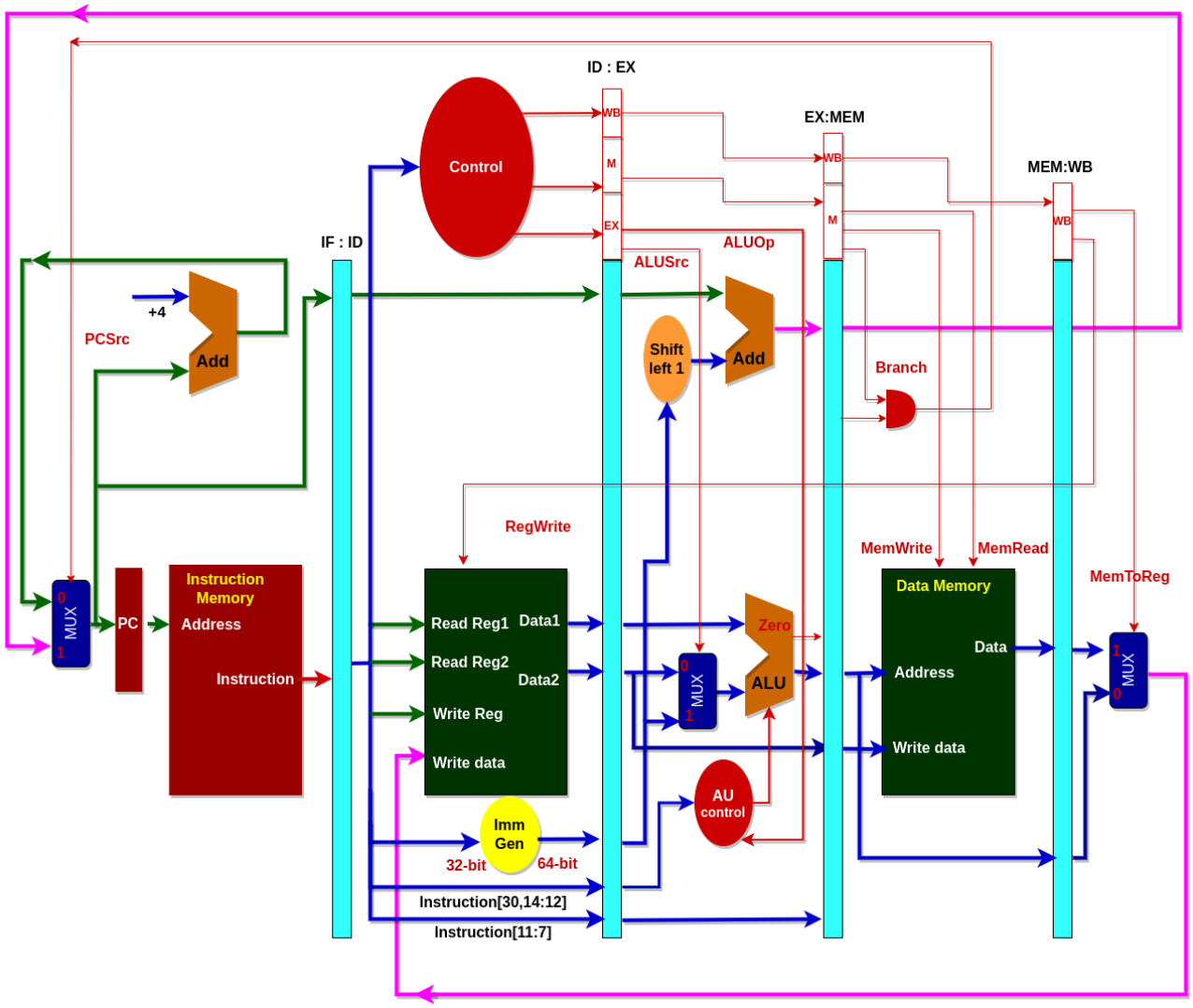


Figure 2.21 The pipelined datapath of **Figure 2.17**, with the control signals connected to the control portions of the pipeline registers.

> The control values for the last three stages are created during the instruction decode stage and then placed in the **ID**/**EX** pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

The examples in this section show the power of pipelined execution and how the hardware performs the task. It's now time to take off the rose-colored glasses and look at what happens with real programs.

In real world the pipelined execution needs to deal with different kinds of hazards.

This topic will be addressed in the next sections, that follow the introduction of **a simple RISC-V model** built with **Verilog** and executing a limited number of **R-type of instructions**.

3. Simple RISC-V Verilog model for R-type instructions

Most of modern implementations of RISC-V are developed with Verilog hardware description and design language. In our case we will try to model a limited part of RISC-V with **R-type** instructions only.

The following figure shows the required components to execute **R-Type** instructions.

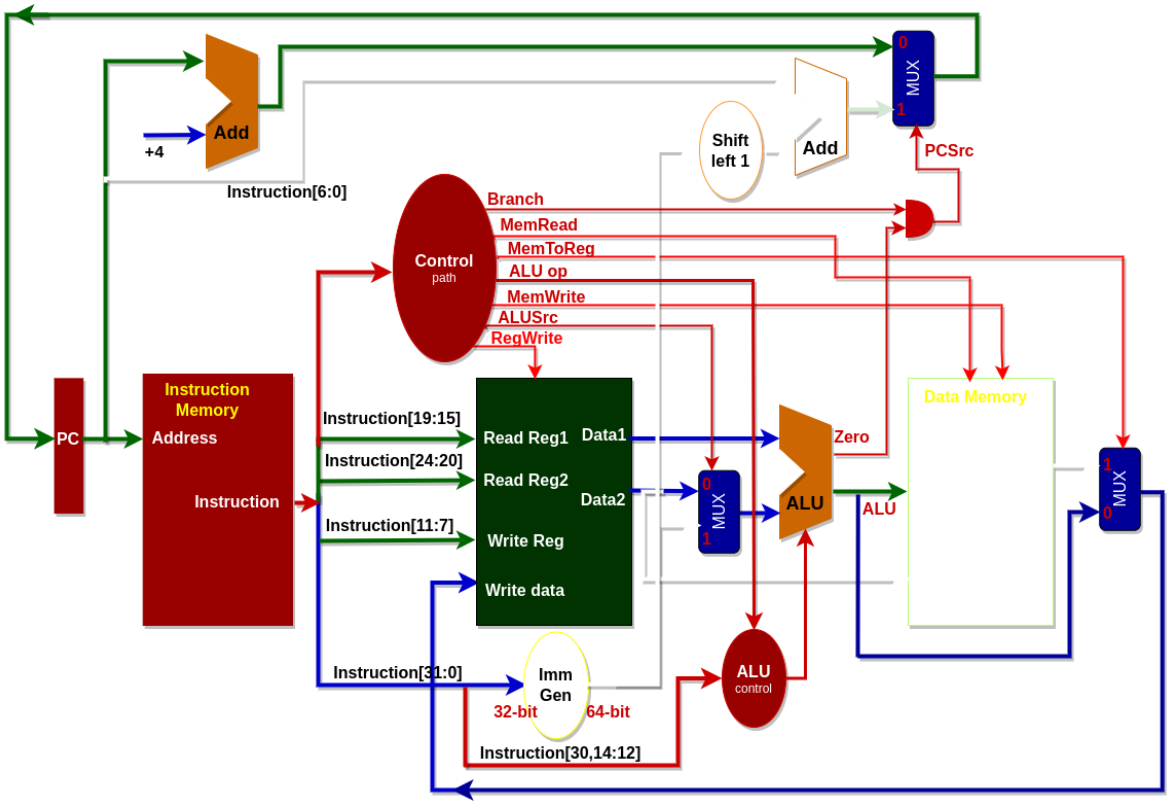


Fig 3.1 The datapath in operation for an **R-type** instruction, such as **`add x1,x2,x3`**. The control lines, datapath units, and connections that are active are highlighted.

The following figure shows the block architecture of the Verilog RISC-V model.

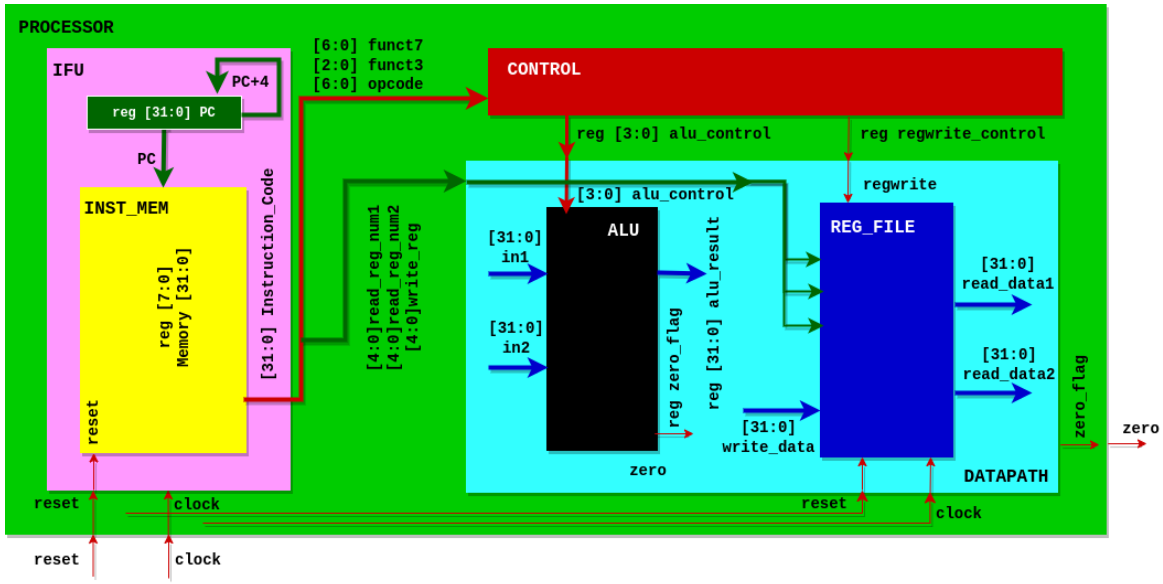


Fig 3.2 Verilog modules for RISC-V architecture for **R-type instructions** (only)

Let us present the modules illustrated in **Figure 3.2.**

The top level **module** of the Verilog code is called **PROCESSOR.v** and it looks like this:

```
`include "CONTROL.v"
`include "DATAPATH.v"
`include "IFU.v"

module PROCESSOR(
    input clock,
    input reset,
    output zero
);

    wire [31:0] instruction_code;
    wire [3:0] alu_control;
    wire regwrite;

    IFU IFU_module(clock, reset, instruction_code);

    CONTROL control_module(instruction_code[31:25],instruction_code[14:12],instruction_code[6:0],
                           alu_control, regwrite);

    DATAPATH datapath_module(instruction_code[19:15], instruction_code[24:20],
                             instruction_code[11:7], alu_control, regwrite, clock, reset, zero);

endmodule
```

The **PROCESSOR** module includes the control module **CONTROL.v**, the datapath module **DATAPATH.v** and the instruction fetch module **IFU.v** .
The module has only **two simple signal inputs**: **clock** and **reset,** and **one output: zero.**
The internal connections are wires to carry the **instruction_code** – **32 bits**, the **alu_control** – **4 bits**, and simple control signal **regwrite**.

The module instantiates 3 sub-modules:

```
IFU IFU_module(clock, reset, instruction_code);
```
that receives clock and reset signals and sends one output signal **instruction_code** on 32 bits.

```
CONTROL control_module(instruction_code[31:25],
                       instruction_code[14:12],instruction_code[6:0],
                       alu_control, regwrite);
```

That receives the instruction code with **funct7 – [31:25]**, **funct3 – [14:12]** and **opcode** fields **– [6:0]**; and sends **alu_control** signal on 4 bits, and simple **regwrite** signal.
Inside the **CONTROL** module , the output signals are registered as:

```
      output reg [3:0] alu_control,
      output reg regwrite_control

DATAPATH datapath_module(instruction_code[19:15], instruction_code[24:20],
                         instruction_code[11:7], alu_control,
                         regwrite, clock, reset, zero);
```

DATAPATH module receives the **addresses of 3 registers**, two input registers: **[11:7]**, **[24:20]** and one output register **[19:15]**. The module also takes **seven control signals**: four at **alu_control** and one at **regwrite**, **clock**, **reset**. The module outputs **one signal**: **zero**.

3.1 Control path

Inside the **IFU** (**Instruction Fetch Unit**) we find the program counter : **reg [31:0] PC** ; and the **INST_MEM** submodule.

3.1.1 Instruction fetch unit - IFU

The following is the Verilog code of **IFU** module:

```
`include "INST_MEM.v"

module IFU(
    input clock,reset,
    output [31:0] Instruction_Code
);
reg [31:0] PC = 32'b0;  // 32-bit program counter is initialized to zero

    // Initializing the instruction memory block
    INST_MEM instr_mem(PC,reset,Instruction_Code);

    always @(posedge clock, posedge reset)
    begin
        if(reset == 1)  //If reset is one, clear the program counter
        PC <= 0;
        else
        PC <= PC+4;   // Increment program counter on positive clock edge
    end

endmodule
```

The instruction fetch unit has **clock** and **reset** pins as input and **32-bit instruction code as output**. Internally the block has Instruction Memory (**INST_MEM instr_mem**), Program Counter (**PC**) and an adder to **increment counter by 4**, on **every positive clock edge**.

```
always @(posedge clock, posedge reset)
```

3.1.2 Instruction Memory unit - INST_MEM

INST_MEM module has two inputs: address line from **PC** on 32 bits, and **reset** signal. It has one output on 32-bits, that carries **Instruction_Code**.

INST_MEM module integrates a modest size of 128 bytes organized in 32 4-byte words.

```
    reg [7:0] Memory [31:0]; // Byte addressable memory with 32 locations

module INST_MEM(
    input [31:0] PC,
    input reset,
    output [31:0] Instruction_Code
);
    reg [7:0] Memory [31:0]; // Byte addressable memory with 32 locations

    // Under normal operation (reset = 0), we assign the instr. code, based on PC
    assign Instruction_Code = {Memory[PC+3],Memory[PC+2],Memory[PC+1],Memory[PC]};

    // Initializing memory when reset is one
    always @(reset)
    begin
        if(reset == 1)
        begin
            // Setting 32-bit instruction: add t1, s0,s1 => 0x00940333
            Memory[3] = 8'h00;
            Memory[2] = 8'h94;
            Memory[1] = 8'h03;
            Memory[0] = 8'h33;
            // Setting 32-bit instruction: sub t2, s2, s3 => 0x413903b3
            Memory[7] = 8'h41;
            Memory[6] = 8'h39;
            Memory[5] = 8'h03;
            Memory[4] = 8'hb3;
            // Setting 32-bit instruction: mul t0, s4, s5 => 0x035a02b3
            Memory[11] = 8'h03;
            Memory[10] = 8'h5a;
```

```
            Memory[9] = 8'h02;
            Memory[8] = 8'hb3;
            // Setting 32-bit instruction: xor t3, s6, s7 => 0x017b4e33
            Memory[15] = 8'h01;
            Memory[14] = 8'h7b;
            Memory[13] = 8'h4e;
            Memory[12] = 8'h33;
            // Setting 32-bit instruction: sll t4, s8, s9
            Memory[19] = 8'h01;
            Memory[18] = 8'h9c;
            Memory[17] = 8'h1e;
            Memory[16] = 8'hb3;
            // Setting 32-bit instruction: srl t5, s10, s11
            Memory[23] = 8'h01;
            Memory[22] = 8'hbd;
            Memory[21] = 8'h5f;
            Memory[20] = 8'h33;
            // Setting 32-bit instruction: and t6, a2, a3
            Memory[27] = 8'h00;
            Memory[26] = 8'hd6;
            Memory[25] = 8'h7f;
            Memory[24] = 8'hb3;
            // Setting 32-bit instruction: or a7, a4, a5
            Memory[31] = 8'h00;
            Memory[30] = 8'hf7;
            Memory[29] = 8'h68;
            Memory[28] = 8'hb3;
        end
    end

endmodule
```

Instruction memory takes in two inputs: a **32-bit Program Counter** and a 1-bit **`reset`**.
The memory is **initialized** when **`reset`** is **`1`**. When reset is set to 0, Based on the value of **`PC`**, corresponding 32-bit Instruction code is output.

3.2 Data path

DATAPATH module is built from two submodules: **REG_FILE** module and **ALU** module. The register address fields - signals are connected to **REG_FILE** address inputs to select the required registers (two input registers, and one output register). The **REG_FILE** has 3 data busses to carry the data from/to selected registers, as well as 3 control signals for: **regwrite**, **clock**, and **reset**.

ALU module inputs are: **[31:0] in1,in2** for input data on 32-bit bus, and the output result on **reg [31:0] alu_result.** Note that the output is **registered. ALU** control is provided via **[3:0] alu_control** lines. The second "result" - output from the ALU module is **zero_flag** signal.

3.2.1 Register file : 32 32-bit registers

A register file can read two registers and write in to one register at the "same" time. The RISC V register file contains total of 32 registers each of size 32-bit. Hence 5-bits are used to specify the register numbers that are to be read or written.
Register file **always outputs** the contents of the register corresponding to read register numbers specified.
Reading a register **is not dependent** on any other signals.
Register writes are controlled by a control signal **RegWrite.** The write should happen if **RegWrite** signal is set to 1 and if there is **positive edge of clock**.

```
module REG_FILE(
    input [4:0] read_reg_num1,
    input [4:0] read_reg_num2,
    input [4:0] write_reg,
    input [31:0] write_data,
    output [31:0] read_data1,
    output [31:0] read_data2,
    input regwrite,
    input clock,
    input reset
);

    reg [31:0] reg_memory [31:0]; // 32 memory locations each 32 bits wide
    integer i=0;

    //  When reset is triggered, we initialize the registers with some values
    always @(posedge reset)
    begin
         reg_memory[0] = 32'h0;
         reg_memory[1] = 32'h1;
         reg_memory[2] = 32'h2;
         reg_memory[3] = 32'h3;
         reg_memory[4] = 32'h4;
         reg_memory[5] = 32'h5;
         reg_memory[6] = 32'h6;
         reg_memory[7] = 32'h7;
         reg_memory[8] = 32'h8;
         reg_memory[9] = 32'h9;
         reg_memory[10] = 32'h10;
         reg_memory[11] = 32'h11;
         reg_memory[12] = 32'h12;
         reg_memory[13] = 32'h13;
         reg_memory[14] = 32'h14;
         reg_memory[15] = 32'h15;
         reg_memory[16] = 32'h16;
         reg_memory[17] = 32'h17;
         reg_memory[18] = 32'h18;
         reg_memory[19] = 32'h19;
         reg_memory[20] = 32'h20;
         reg_memory[21] = 32'h21;
         reg_memory[22] = 32'h22;
         reg_memory[23] = 32'h23;
         reg_memory[24] = 32'h24;
         reg_memory[25] = 32'h25;
         reg_memory[26] = 32'h26;
         reg_memory[27] = 32'h27;
         reg_memory[28] = 32'h28;
         reg_memory[29] = 32'h29;
         reg_memory[30] = 32'h30;
         reg_memory[31] = 32'h31;
    end
```

```
    assign read_data1 = reg_memory[read_reg_num1];
    assign read_data2 = reg_memory[read_reg_num2];

    // If clock edge is positive and regwrite is 1, we write data to specified register
    always @(posedge clock)
    begin
        if (regwrite) begin
            reg_memory[write_reg] = write_data;
        end
    end

endmodule
```

3.2.2 ALU unit

ALU module takes two operands of size 32-bits each and a 4-bit **alu_control** as input. Operation is performed on the basis of **alu_control** value and output is 32-bit **ALU_result**. If the **alu_result** is zero, a **zero_flag** is set.

ALU Control lines (**alu_control**) operation:

```
0000      Bitwise-AND
0001      Bitwise-OR
0010      Add (A+B)
0100      Subtract (A-B)
1000      Set on less than
0011      Shift left logical
0101      Shift right logical
0110      Multiply
0111      Bitwise-XOR
```

```
module ALU (
    input [31:0] in1,in2,
    input[3:0] alu_control,
    output reg [31:0] alu_result,
    output reg zero_flag
);
    always @(*)
    begin
        // Operating based on control input
        case(alu_control)
        4'b0000: alu_result = in1&in2;
        4'b0001: alu_result = in1|in2;
        4'b0010: alu_result = in1+in2;
        4'b0100: alu_result = in1-in2;
        4'b1000: begin
            if(in1<in2)
            alu_result = 1;
            else
            alu_result = 0;
        end
        4'b0011: alu_result = in1<<in2;
        4'b0101: alu_result = in1>>in2;
        4'b0110: alu_result = in1*in2;
        4'b0111: alu_result = in1^in2;

        endcase

        // Setting Zero_flag if ALU_result is zero
        if (alu_result == 0)
            zero_flag = 1'b1;
        else
            zero_flag = 1'b0;

    end
endmodule
```

3.3 Processor test

The test of the **PROCESSOR** module requires the preparation of test bench module. The test bench module integrates the **PROCESSOR** module and provides the stimuli to animate it.
The stimuli signals are mainly reset and clock.

Initially **reset** prepares the starting time (here **#50** time units – nanoseconds). The second **initial** block starts the generation of the clock signal that has 2*20 nanosecond cycle (**#20 clock = ~clock**)and turns **forever.**

Note that the initial **initial** block prepares the name of the **trace file** - **dumpfile** and indicates the names off variables/signals to be registered. In case of first argument = **0,** all variables are to be traced.

```
`include "PROCESSOR.v"

module stimulus ();

    reg clock;
    reg reset;
    wire zero;

    // Instantiating the processor!!!
    PROCESSOR test_processor(clock,reset,zero);

    initial begin
        $dumpfile("output_wave.vcd");
        $dumpvars(0,stimulus);
    end

    initial begin
        reset = 1;
        #50 reset = 0;
    end

    initial begin
        clock = 0;
        forever #20 clock = ~clock;
    end

    initial
    #300 $finish;

endmodule
```

Let us call register the above code in **Processor_tb.v** file.

Now we have all components to start the **compilation** and **simulation of the compiled model**.

If you have previously installed the (Icarus Verilog – **iverilog**, and **GTKWave** tool) compiler/simulator you can **compile** the model (having all modules in the same directory) with:

```
iverilog Processor_tb.v  -o  proc-compiled
```

After the compilation you can **execute** the compiled model with:

```
vvp proc-compiled
```

The waveform generated from **testbench** is named as **output_wave.vcd**

```
gtkwave output_wave.vcd
```

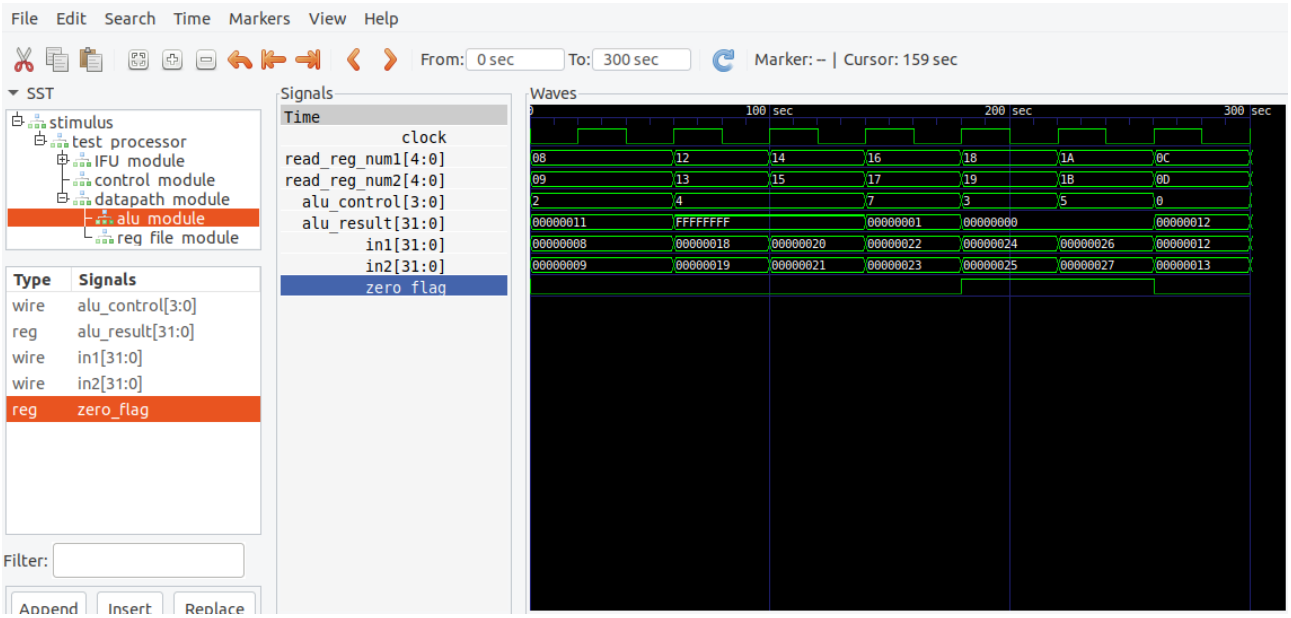


Fig 3.3 GTKWave graph with ALU module signals for the given model execution.

You can trace the **`alu_result`** for operation **`alu_oper = 4`** meaning **subtraction** (look in the module **`ALU`**) and two input values: **`in1=18`**, **`in2=19`**, **what is the result ?**

To do:

1. Analyze the “given” result
2. Modify the Instruction Memory content using “your” program – instruction sequence and test it.

Table of Contents