Imperial College London

ISA and Compilers: Year 2

Processor: Control and Data Path

Xin Wang

October 11, 2020

Abstract

Unlike other instruction architectures like the Mu0, the ALU is used in almost every instruction in the MIPS architecture from common arithmetic to address calculation.

MIPS architecture supports negative and positive arithmetic as well as floating numbers i.e. decimals so it is important to understand how the ALU enables those features.

Contents

1 The basic MIPS implementation

From Year 1 DECA, the following concepts are already covered:

- How to split instruction into opcode and operands.
- How to encode instructions as bits.
- How to build an ALU.

This chapter focuses on how to build a datapath and design a suitable control path. To aid in this process, an implementation that including a subset of the core MIPS instruction set:

- **Memory-access instructions**: load word (`lw`) and store word (`sw`)
- **Register-based instructions**: `add`, `sub`, `AND`, `OR`, and `slt`
- **Branch instructions**: branch equal (`beq`) and jump (`j`)

is examined. These instructions illustrates the key principles used in creating a datapath and designing the control. The implementation of the remaining instructions is similar.

The importance of this chapter is to notice how the instruction set architecture determines many aspects of the implementation, and how the choice of various implementation strategies affects the clock rate and CPI for the computer. Many of the key design principles introduced in earlier can be illustrated by looking at the implementation.

1.1 Implementation overview

Much of the implementations of these instructions is the same, independent of the exact class of the instruction. For every instruction, the first two steps are identical:

1. Send the Program Counter (PC) content to the Memory which contains the code and fetches the instruction from the Memory.
2. Read one or two registers, using fields of the instruction to select which of the registers to read e.g. for the `lw` instruction, only one register is read, but other instructions like `add` would require reading two registers.

After these two steps, the actions required to complete the instruction depend on the instruction classes. For each of the three instruction classes the actions are largely the same - independent of the exact instruction. The simplistic and regularity of the MIPS instruction set results in making the execution of many of the instruction classes similar.

A high-level view of a MIPS implementation is given, emphasising on the **various functional units** and **interconnection fo function units**.

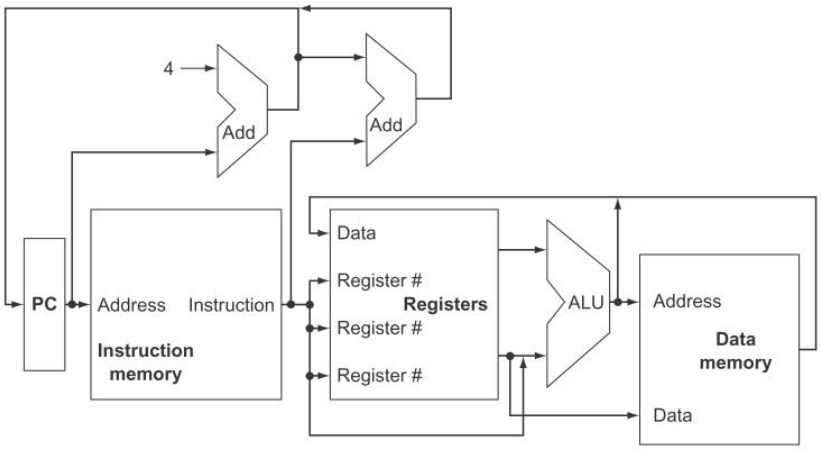


Figure 1: A high-level view of a MIPS implementation

Data process:

1. All instructions start with using the *PC* to supply the **instruction address** to the *Memory* containing instructions and data.

2. After the instruction is fetched, the fields of that instruction is processed and determines the register operands to be used.

3. Required register operands are fetched from Memory.

4. Once the register operands are fetched, it is operated on differently by the ALU:

   - To compute a memory address (for 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.

   - To compute an arithmetic result (for an integer arithmetic-logical instruction)

     The result from the ALU must be written to a register.

   - To compare (for a branch)

     Branches require the use of the ALU output to determine the next instruction address, which comes either from the ALU (where the PC and branch off set are summed) or from an adder that increments the current PC by 4.

This view shows most of the flow of data through the processor but it omits two important aspects of instruction execution:

1. Note in several places, the diagram shows data going into a particular

unit as coming from two different sources merging into one e.g. the data coming into the PC is a merger from the two Add blocks.

Practically, these data lines are not wired together. A logic element is added that chooses from the multiple sources and selects one source only based on the setting of its control lines. This logic element is called a **multiplexor** or **data selector**.

2. Several of the units must be controlled depending on the type of instruction e.g. data memory must read on a load and written on a store. Register files must be written only on a load or an arithmetic-logical instruction. Control lines that are set depending on the basis of the various fields in the instruction control these operations.

The final overview diagram shows the datapath of the previous with the three required multiplexors added and the control lines for the major functional units.

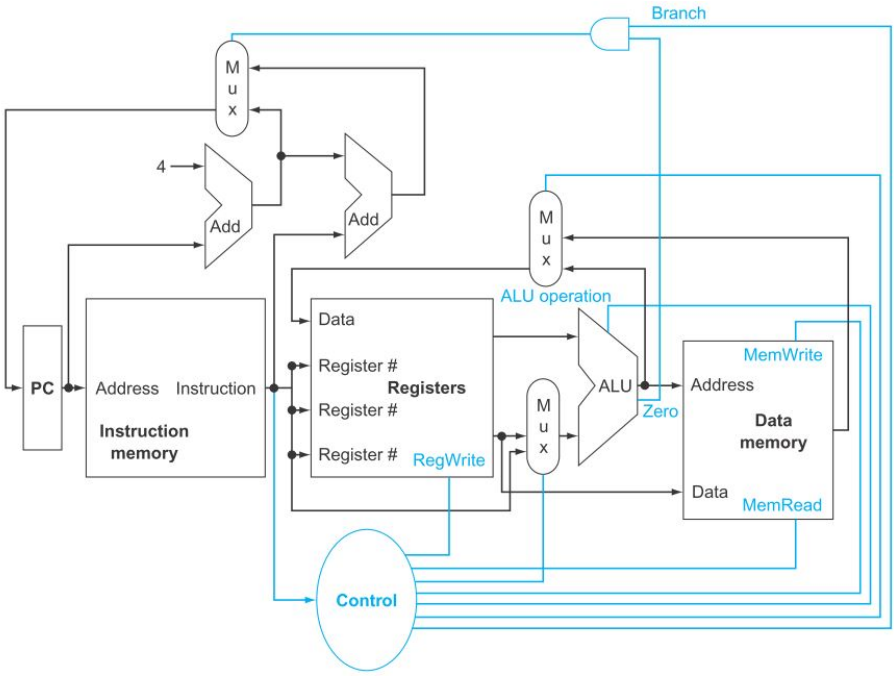


Figure 2: Implementation of MIPS subset including necessary multiplexors and control lines.

1. **Control Unit** has the *instruction* as an input and controls the *control lines* for the functional units and two of the multiplexors.

2. **Top multiplexor** chooses between whether $PC + 4$ or the branch destination address based on the Zero output of the ALU i.e. performs the comparison of a `beq` instruction.

3. **Middle multiplexor** used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the *data memory* (in the case of a load) into the register file.

4. **Bottom multiplexor** used to determine if 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 regularity and simplicity of the MIPS instruction set allows a simple decoding process to be used to determine how to set the control lines.

2 Logic design conventions

When implementing the hardware logic of a computer, how it operates and how the computer is clocked are important aspects to consider.

> **Combinational**: The output of the element only depends on the current inputs. Common combinational elements are AND gate or ALU.
>
> **State element**: An element some internal storage i.e. a memory element. Common state elements are flip-flops, registers and memory.

The datapath in MIPS consist of two different types of logic elements:

- Elements that operate on data values:

  The elements are all **combinational**. Given the same input, a combinational element always produces the same output because it has no internal storage.

- Elements that contain **state**:

  State elements are important elements as it characterises the computer. A state element has **at least two inputs**:

  1. The data value to be written into the element
  2. The clock that determines when the data value can be written. A state element can be read at any time.

  and one output that provides the value that **was written in an earlier clock cycle**.

The opposite of **combinational** is **sequential**. Logic components that contain state are sequential since the outputs depend on both **its inputs** and **the contents of the internal state**.

2.1 Clocking

Clocking methodology: The approach used to determine when data is valid and stable relative to the clock.

A clocking methodology is required to make hardware predictable by specifying the timing of reads and writes. It prevents the situation where reading and writing occurs simultaneously. This would result ina glitch - unpredictable behaviour.

The easiest of clocking methodology is **edge-triggered methodology**. There are two types: **positive-edge triggered** and **negative-edge triggered**.

Edge-triggered clocking: A clocking scheme in which all state changes occur on a clock edge. Any values stored in a sequential logic element are updated only on a clock edge

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.

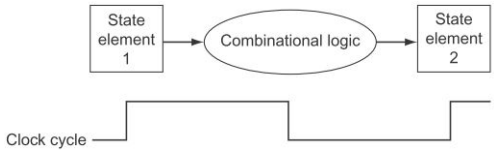


Figure 3: Relation between combinational logic, state elements and the clock.

Single clock cycle - Time taken for all signals to propagate from state element 1, through the combinational logic, and to state element 2.

Control signal: A signal used for multiplexor selection or for directing the operation of a functional unit.

Data signal: A signal that contains information that is operated on by a functional unit.

A state element is either written on **every clock edge** or controlled by **an explicit control signal**. The state element is changed **only when the write control signal is asserted and a clock edge occurs**.

3 Building the Data Path

> **Datapath element**: A unit used to operate on or hold data within a processor. In the MIPS implementation, the datapath elements include the instruction and data memories, the register file, the ALU, and adders.

The three-stage compositional approach:

1. Create each datapath independently by examining the major components required to execute each class of MIPS instructions.
2. Combine datapaths, sharing whenever possible.
3. Add control to dynamically select the correct datapaths depending on the instructions.
    - Data-independent: i.e. add and load
    - Data-dependent: i.e. branches

The fundamental components of any processor design is the following:

1. PC: Fetch the instruction from memory.
2. Memory: Contains data and instructions.
3. Adder: To form the ALU.

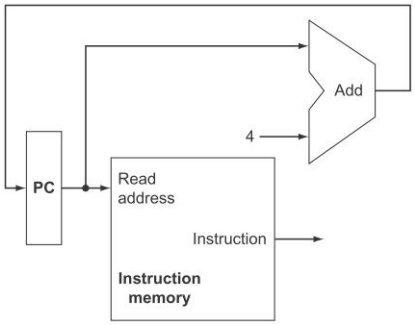


Figure 4: The core portion of the datapath for fetching instructions and incrementing PC.

3.1 Register (R)-based instructions

R-type or arithmetic-logical instructions e.g. `add`, `sub`, `AND` and `slt`. The instructions **read** two registers, **perform** an ALU operation on the contents of the registers, and **write** the result to a register.

Register file: A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed. The processor's 32 general-purpose registers are stored in it.

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
0	17	18	8	0	32
Opcode	Source 1	Source 2	Dest.	Shift	Function

Three register operands: 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: 1) an input to *register file* specifying the *register number* to be read, 2) an output from the *register file* that will carry the value read from the *registers*.
- Writing a data word, two inputs needed: 1) Specify the register number to be written to, 2) Supply the data to be written into the register.

  Process is controlled by the write control signal, which must be asserted for a write to occur at the clock edge.

Figure 5: Register file: The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide.

Example 1: `add $5, $6, $7 #reg[5] = reg[6] + reg[7]`

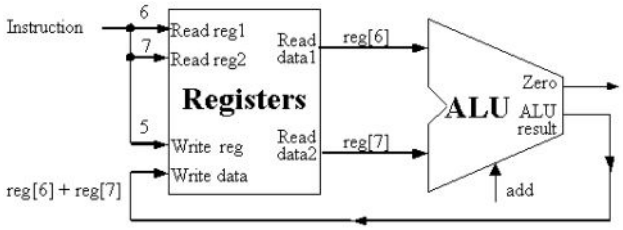


Figure 6: Data path flow of R type instructions

3.2 Memory access (I) type instructions

6 bits	5 bits	5 bits	16 bits
35	19	8	Astart
Opcode	Source	Dest.	Immediate constant

These instructions compute a memory address by adding the base register `$t2`, to the 16-bit signed off-set field contained in the instruction `$t1`. Consider the MIPS instructions:

- *load* word instruction: `lw $t1, offset_value($t2)`

  Value read from memory must be **written into** the *register file* in the specified register `$t1`.

- *store* word instruction: `sw $t1, offset_value($t2)`

  Value to be stored must also be **read from** the *register file* where it resides in `$t1`.

This type of instructions will require the following components:

- *ALU*: Calculate the offset address value.
- *Memory*: Store data and instruction.
- *Sign extend*: Extend the 16-bit off-set field **of the instruction** to a 32-bit signed value to be added on by the *ALU*.

Example 1: `lw $5, offset($6) #reg[5] = M[reg[6] + offset]`

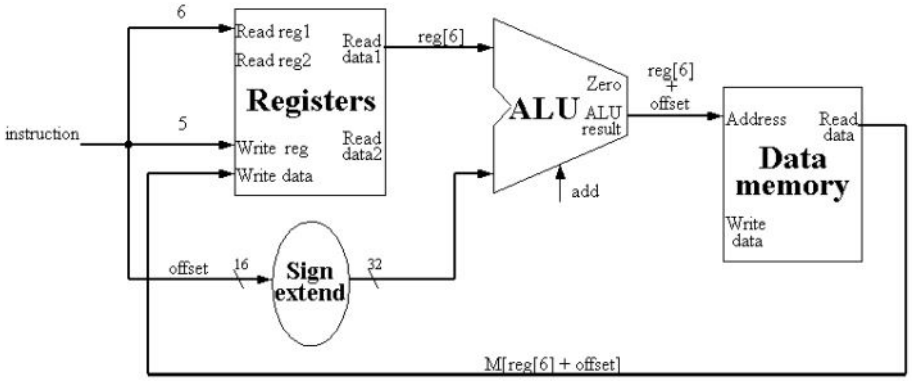


Figure 7: Data path flow of I type instruction `lw`

Example 2: `sw $5, offset($6) # M[reg[6] + offset] = reg[5]`

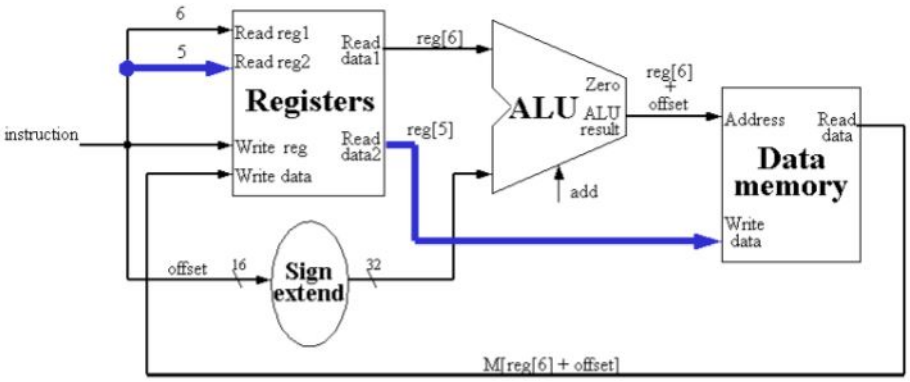


Figure 8: Data path flow of I type instruction `rw`

Branch instructions `beq` are part of I-type instructions.

```
beq  $t1,$t2,offset.
```

The `beq` instruction has three operands:

- Two registers that are used for comparing for equality.
- A 16-bit off-set used to compute the **branch target address** relative to the branch instruction address.

> **branch target address** = The address specified in a branch, which becomes the new *PC* value if the branch is taken. In the MIPS architecture the branch target is given by:
>
> - Sum of the off-set field of the instruction
> - Address of the instruction following the branch

To implement this instruction, the branch target address must be computed by adding the **sign-extended off-set field** of the instruction to the *PC*. There are two important aspects to remember about the MIPS architecture:

1. The MIPS instruction set architecture requires that the **base address used in the branch address calculation** is the **address of the instruction following the branch**.

   Remember the address of the next instruction is simply `PC + 4`, so it is easy to use this value as the base for computing the branch target address.

2. The MIPS architecture also states that **the off-set field is shifted left 2 bits** i.e. it is **a word off-set**. This shift increases the effective range of the off-set field by a factor of 4.

   To deal with that complication, we will need to shift the off-set field by 2.

As well as computing the branch target address, the processor must determine whether the next instruction is:

- The instruction that follows sequentially:

  If the operands are not equal i.e. conditions are not true, the incremented `PC` value will replace the current `PC` value as normal.

- The instruction at the branch target address:

  The operands are equal i.e. conditions is true, the branch target address becomes the new `PC` value.

SO in summary the branch datapath must do **two operations**:

- Compute the branch target address.
- Compare the register contents.

Note: Branches affect the **instruction Fetch portion** of the datapath.

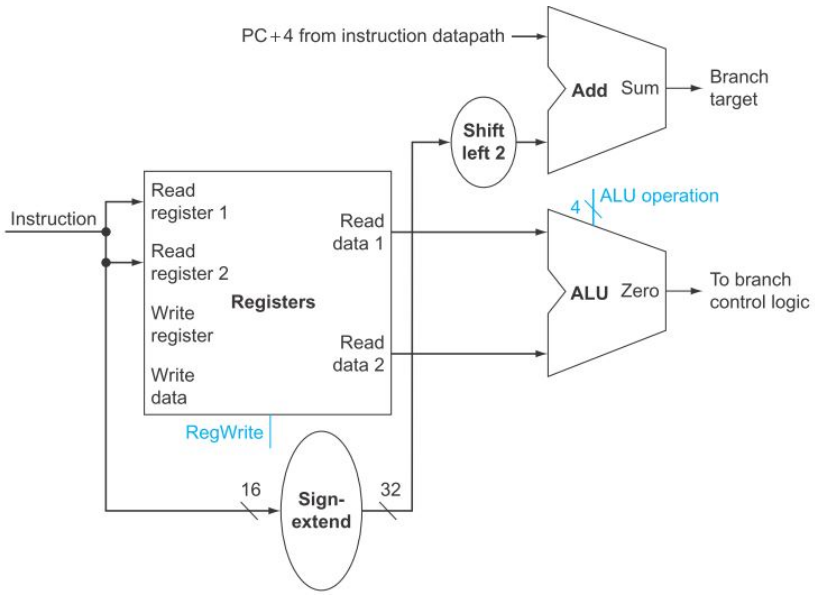


Figure 9: Datapath for a branch using the `ALU` to evaluate the branch condition and a separate adder `Add` to compute the branch target as the sum of the incremented `PC` and the **sign-extended, lower 16 bits** of the instruction, shifted left 2 bits. The control logic from the `ALU` is used to decide whether the incremented PC or branch target should replace the PC.

3.3 Combining into single data path

Now a simple datapath for the core MIPS architecture is formed by adding the two datapath designs.

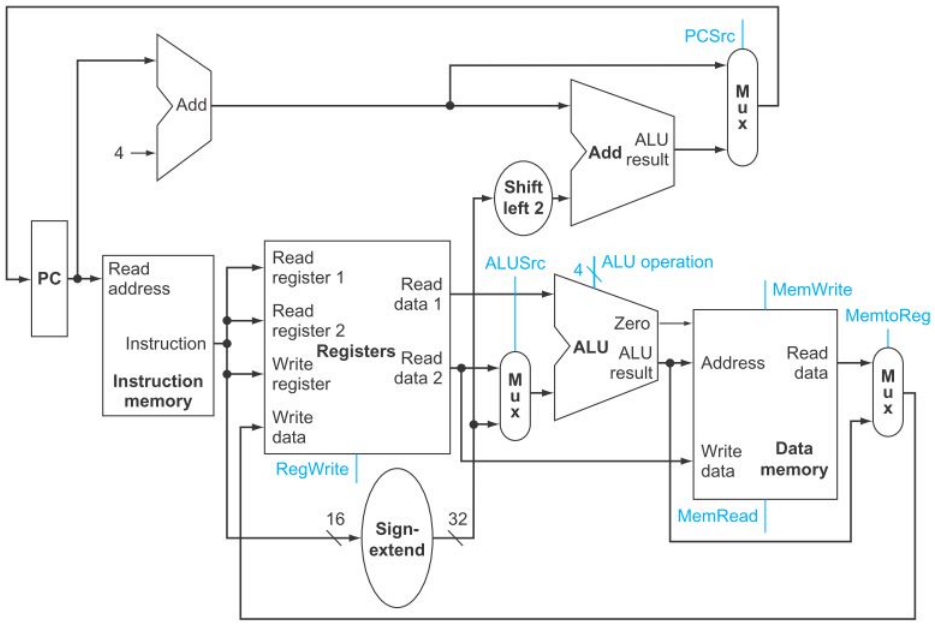


Figure 10: The datapath for the memory instructions and the R-type instructions.

This datapath can execute the basic instructions e.g. load-store word, ALU operations, and branches in a single clock cycle. Just one additional multiplexor is needed to integrate branches.

Given this simple datapath, a control unit is needed:

- Take inputs and generate a write signal for each state element.
- Acts as the selector control for each multiplexor.
- Control the `ALU`.

4 A Simple Implementation Scheme

Implementation of a MIPS subset includes designing **a data path** and **the suitable control functions**.

This simple implementation uses the datapath of the last section and a simple control function. It supports `lw`, `sw`, `beq`, and arithmetic-logical instructions `add`, `sub`, `AND`, `OR`, and `set on less than`.

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

Figure 11: How ALU control bits are set depending on the `ALUOp` control bits and the different function field for the R-type instruction.

4.1 ALU control

In MIPS, depending on the instruction class, the ALU will need to perform one of six functions.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

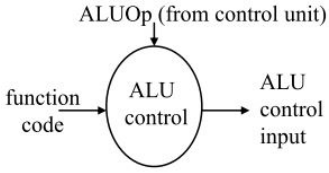
For example:

- `lw` and `sw` instructions, the ALU computes the memory address by addition
- R-type instructions, ALU needs to perform one of the five actions (AND, OR, subtract, add, or set on less than) depending on the value of the 6-bit `funct field`

The **4-bit ALU control input** is set using a control unit with two inputs:

- The function field of the instruction
- `ALUOp`: A 2-bit control field indicating whether the operation to be performed should be:
  - 00: Loads and Stores
  - 01: Subtract for `beq`
  - 10: Determined by the operation encoded in the `funct field`

    Note: The function field **is only used** when the `ALUOp` bits equal 10, a small piece of logic that recognizes the subset of possible values is used.



The output of the `ALU control unit` is a 4-bit signal directly controlling the `ALU` by generating one of the 4-bit combinations shown above.

This design uses multiple levels of decoding i.e. the `main control unit` generates the `ALUOp` bits, which then are used as input to the `ALU control unit` that generates the actual signals to control the `ALU`. This is a common implementation technique because using multiple levels of control can:

- Reduce the size of the main control unit
- Potentially increase the speed of the control unit

These optimizations are important because the speed of the control unit is critical to the clock cycle time.

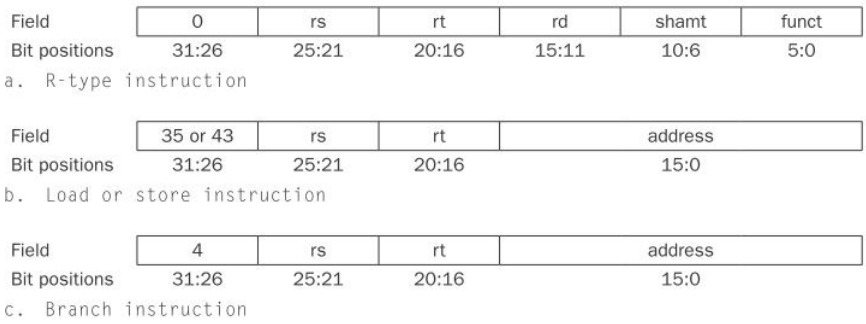
4.2 Designing the Main Control Unit

Identify the fields of an instruction and the control lines that are needed for the datapath constructed above:

ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

Figure 12: The truth table for the 4 ALU control bits

To see the relationship of the fields of an instruction and the datapath, view the formats of the three instruction classes: the R-type, branch, and load-store.



Opcode: The field that denotes the operation and format of an instruction.

Note: The design principle *simplicity favors regularity* is seen here in specifying control. There are several major observations about these instruction formats:

- The `opcode` i.e. **Op[5:0]** is always contained in bits [31 : 26].
- The two registers to be read are always specified by the `rs` at [25 : 21] and `rt` at [20 : 16].
- The 16-bit off set for branch equal, load, and store is always in positions [15 : 0].
- The destination register is in one of two places:
  - For load it is in bit positions [20 : 16]

– For R-type instruction it is in bit positions [15 : 11]

Thus, a multiplexor is needed to select which field of the instruction is used to decide between the two positions.

Using these information, the instruction labels and extra multiplexor is added to the simple datapath.

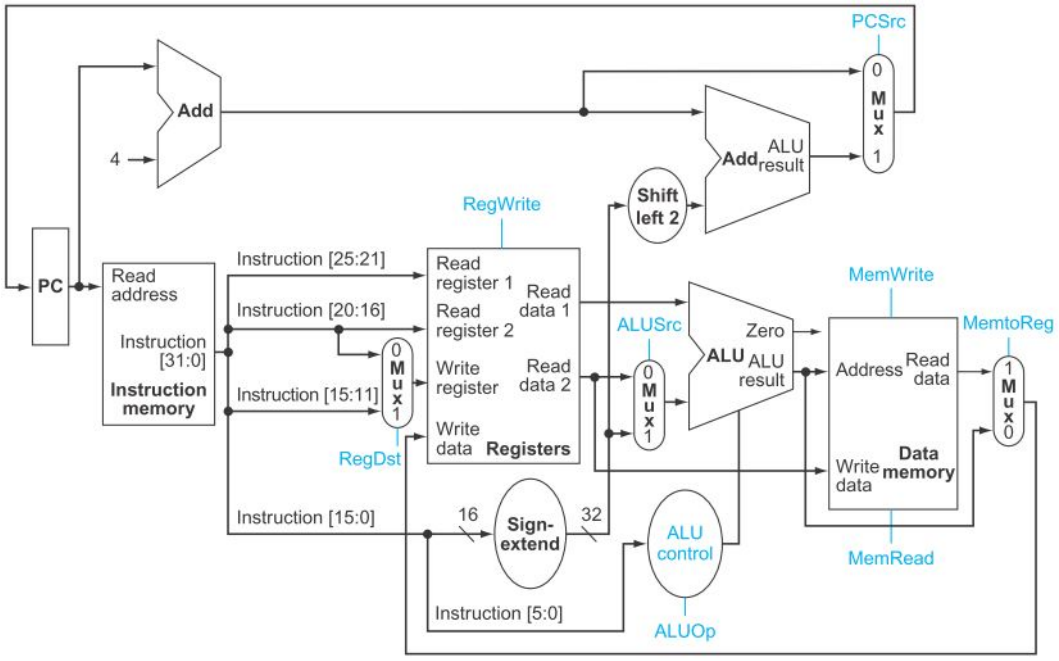


Figure 13: Datapath with all necessary multiplexors and all control lines.

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Figure 14: The control lines determined by the *OpCode* fields of the instruction.

Next are the **seven single-bit control lines plus the 2-bit ALUOp control signal** where the `ALUOp control signal` is used to define what the seven other control signals do.

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
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 (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Figure 15: The effect of each of the seven control signals.

These nine control signals (seven from above and two for ALUOp) is determined by the six input signals to the control unit i.e. the `Opcode` bits [31 : 26].

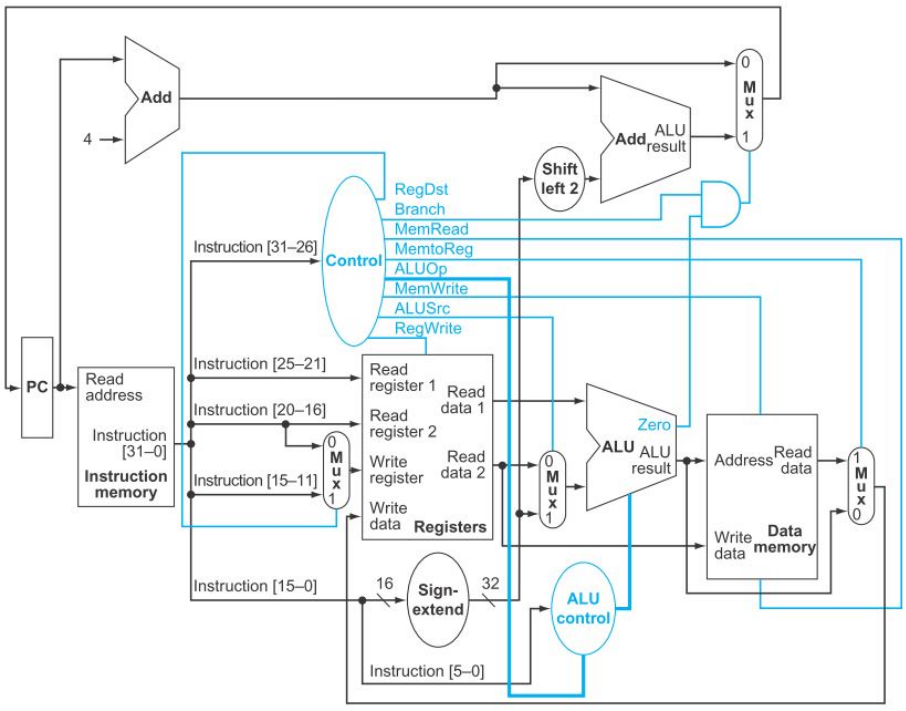


Figure 16: The simple datapath with the control unit

4.3 Operation of the data path

Based on the previous chapters, the information can be used to design the **control unit logic**. To give a short summary, viewing how the three different instruction classes uses the datapath.

The operation of the datapath for an **R-type instruction** e.g. `add $t1,$t2,$t3`. Everything occurs in **one clock cycle** and consists of the four steps that are ordered by the flow of information:

1. An instruction is fetched from *Memory* and the *PC* is incremented.
2. - Two registers `$t2` and `$t3` are read from the *Register File*.
   - The main control unit **computes and sets** the control lines.
3. *ALU* processes the data read from the *Register File* using the function code i.e. instruction funct field $[5:0]$ from the *ALU Control* to output the *ALU* control code.
4. Result from the *ALU* is written into the *Register File* using instruction bits $[15:11]$ to select the destination register `$t1`.

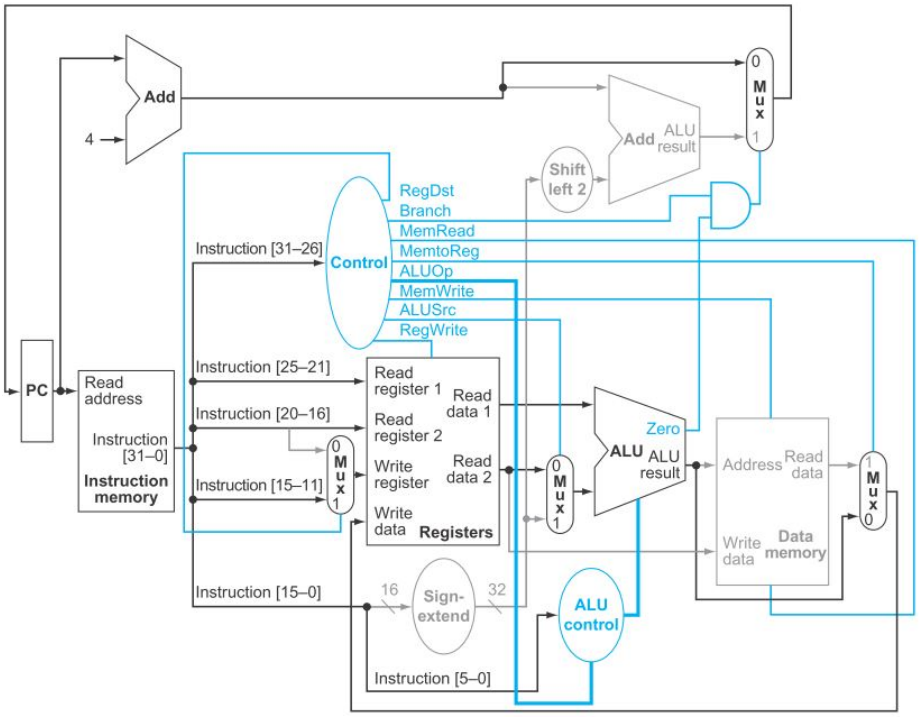


Figure 17: The datapath in operation for an R-type instruction, such as `add $t1,$t2,$t3`.

The execution of a load word instruction e.g. `lw $t1, offset($t2)` can be shown in 5 steps:

1. An instruction is fetched from *Memory* and the *PC* is incremented.
2. Register `$t2` value is read from the *Register File*.
3. *ALU* computes the sum of the value read from: *Register File* and the **sign-extended, lower 16 bits, offseted instruction**.
4. The sum from the *ALU* is used as the address for the *Data Memory*.
5. - Data from *Memory* is written into the *Register File*.
   - The register destination is given by instruction bits $[20:16]$ in register `$t1`.

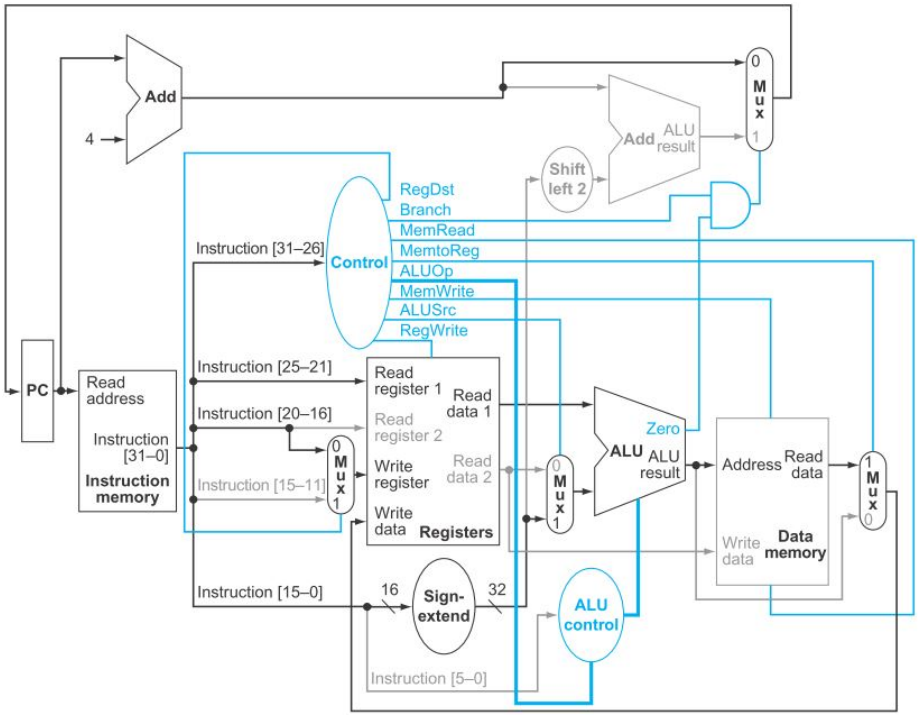


Figure 18: The datapath in operation for a load instruction.

The execution of branch-on-equal instruction e.g. `beq $t1, $t2, 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 **branch target address**. It can be shown in four steps:

1. An instruction is fetched from *Memory* and the *PC* is incremented.
2. Two registers `$t2` and `$t3` are read from the *Register File*.
3. - The *ALU* perform subtraction `sub` on data values from the *Register File*.
   - The value **PC + 4** is added to the **sign-extended, lower 16 bits, offset left by 2 bits instruction**.
   - The result is the branch target address.
4. The **Zero result** from the *ALU* is used to decide which adder result to store into the *PC*.

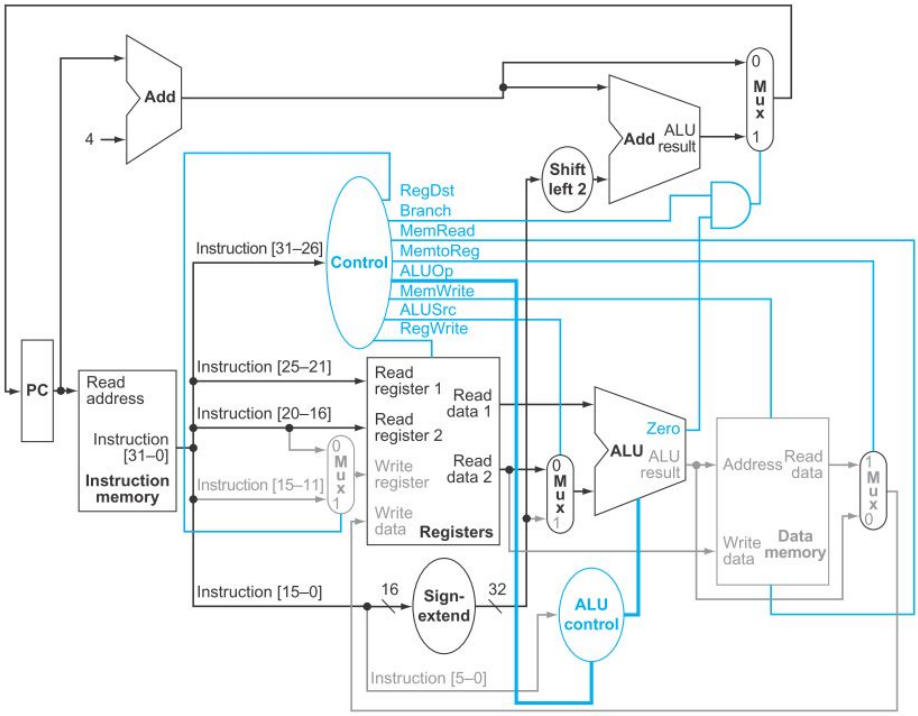


Figure 19: The datapath in operation for a branch-on-equal instruction. Note: After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.

4.4 Finalising the Control

The control function is defined using the contents of the following table:

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

- **Input**: The 6-bit *OpCode* field: Op[5 : 0]
- **Outputs**: The control lines

A truth table for each of the outputs based on the binary encoding of the *OpCodes* can be produced:

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Figure 20: The control function for the simple single-cycle implementation is completely specified by this truth table.

A **single-cycle implementation** of most of the MIPS core instruction set has been implemented.

> **Single-cycle implementation** or **Single clock cycle implementation**: An implementation in which an instruction is executed in one clock cycle. Easy to understand but too slow to be practical.

4.5 Jump instructions

The final step is to add jump instructions to the basic datapath and control unit.

The jump instruction is similar to a branch instruction but has the following differences:

- computes the target PC differently
- is not conditional

Figure 21: Instruction format for the jump instruction (opcode = 2)

Calculating the 32-bit binary jump address:

- Like branch instructions, the low-order (right) 2 bits of the binary jump address are 00.
- The next lower 26 bits of this 32-bit address come from the 26-bit immediate field in the jump instruction.
- The upper 4 bits of the address that should replace the PC come from the PC of the jump instruction plus 4.

This means a jump is implemented by storing into the PC the concatenation of:

1. Upper 4 bits of the current $PC + 4$ i.e. bits $[31 : 28]$ of the sequentially following instruction address.
2. The 26-bit immediate field of the jump instruction.
3. The bits 00_{two}

To add the jump feature into the existing data path design, the following changes are made:

1. An additional multiplexor used to select the source for the new PC value which is either:
   - The incremented PC i.e. $PC + 4$
   - The branch target PC (Supports `beq`)
   - The jump target PC
2. One additional control signal required for the additional multiplexor called **Jump**. **Jump** is only asserted when instruction is a jump instruction i.e. *OpCode* is 2

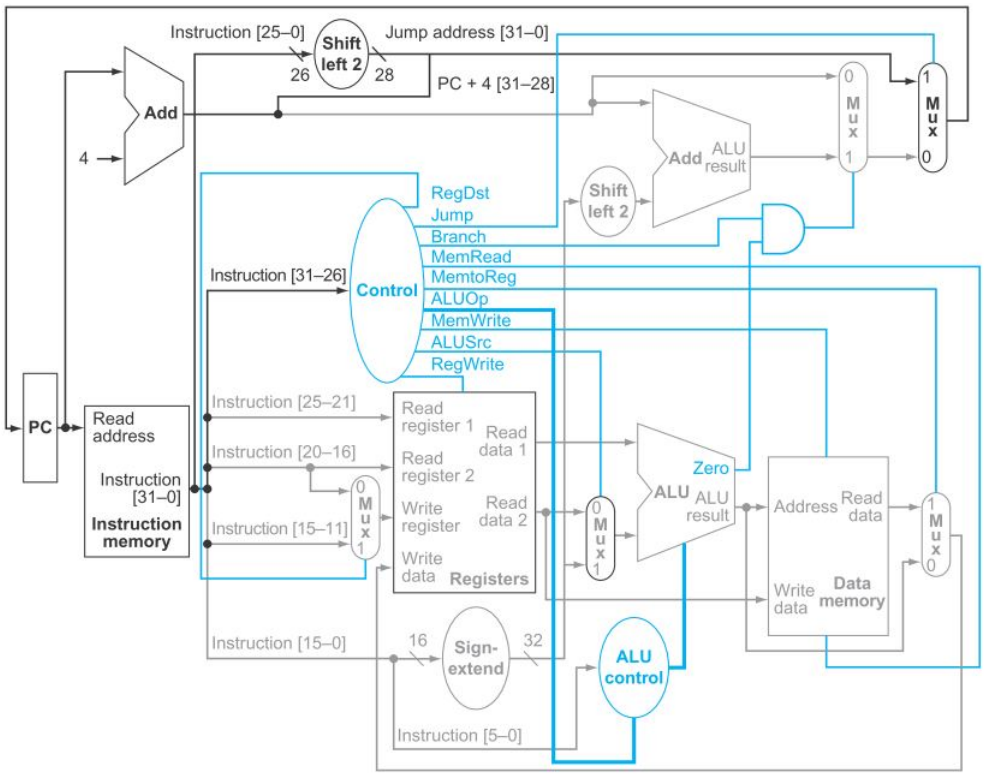


Figure 22: The simple control and datapath are extended to handle the jump instruction

5 Single-cycle implementation

Single-cycle designs are not used in modern designs because it is inefficient.

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**. In the MIPS instruction set, this path is a load instruction - five functional units are used in series:

- the Instruction Memory
- the Register File
- the ALU
- the Data Memory
- the Register File

The penalty for using the single-cycle design with a fixed clock cycle is significant but it is acceptable for small instruction sets.