Chapter 9
Data Paths and Instruction Execution

Douglas Comer. (2017). *Essentials of Computer Architecture* (2nd ed.). CRC Press.

M. Morris Mano, Charles R. Kime. (2015). *Logic and computer design fundamentals* (5th ed.). Pearson.

Contents

Contents

1. Data Paths

- At a high level, we are only interested in how instructions are read from memory and how an instruction is executed.
- We consider the addition operation, we will see the data paths across which two operands travel to reach an *Arithmetic Logic Unit (ALU)* and a data path that carries the result to another unit.
- To make the discussion of data paths clear, we will examine a simplified computer. The simplifications include:
  - Our instruction set only contains four instructions
  - We assume a program has already been loaded into memory
  - We ignore startup and assume the processor is running
  - We assume each data item and each instruction occupies exactly 32 bits
  - We only consider integer arithmetic
  - We completely ignore error conditions, such as arithmetic overflow

2. The Example Instruction Set

- A new computer design must begin with the design of an instruction set. Once the details of instructions have been specified, a computer architect can design hardware that performs each of the instructions.
- We will consider an imaginary computer that has the following properties:
  - A set of sixteen general-purpose registers (32 bit registers)
  - A memory that holds instructions (i.e., a program)
  - A separate memory that holds data items

2. The Example Instruction Set

- Four basic instructions that our imaginary computer implements

Instruction	Meaning
add	Add the integers in two registers and place the result in a third register
load	Load an integer from the data memory into a register
store	Store the integer in a register into the data memory
jump	Jump to a new location in the instruction memory

Figure 6.1 Four example instructions, the operands each uses, and the meaning of the instruction.

add r4, r2, r3 : Add the contents of r2 and r3 and placing the result in r4

load r1, 20(r3) : Load the value from locarion (r3 + 20) into r1. 20 is the offset.

jump 60(r11) : The next location where an instruction is execute is (r11 + 60). 60 is the offset.

3. Instructions in Memory

- The instruction format for our imaginary computer.
  - Each instruction has exactly the same format, even though some of the fields are not needed in some instructions. A uniform format makes it easy to design hardware that extracts the fields from an instruction.
- The *operation* (opcode) field in an instruction value that specifies the operation.
  - add: 1, load: 2, store: 3, jump: 4

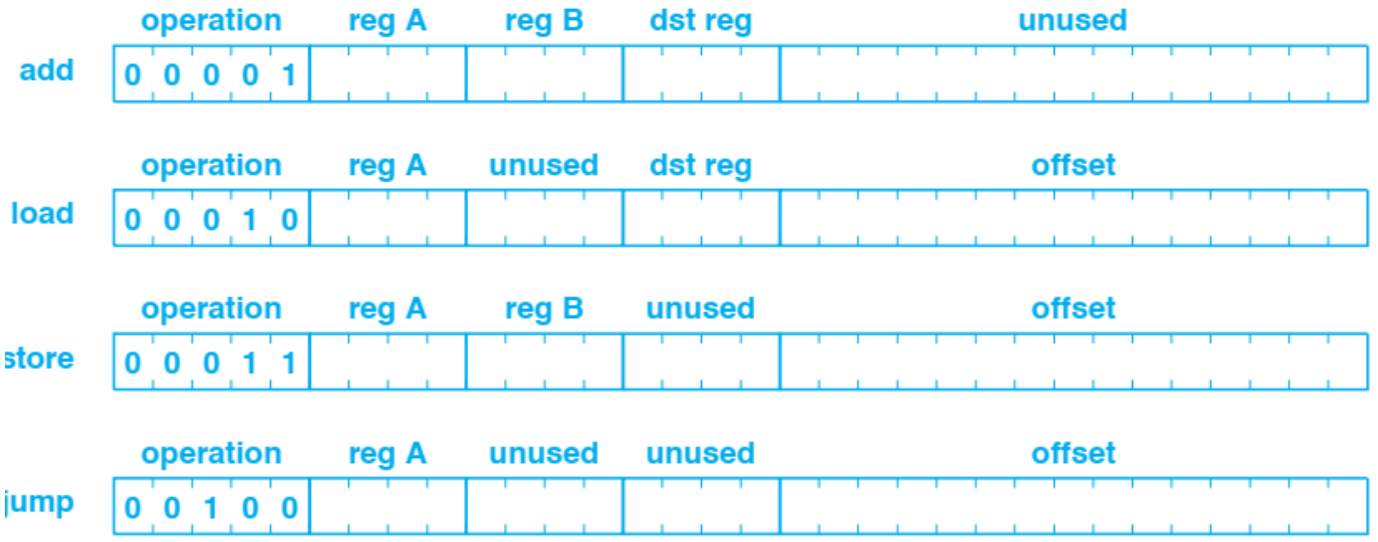


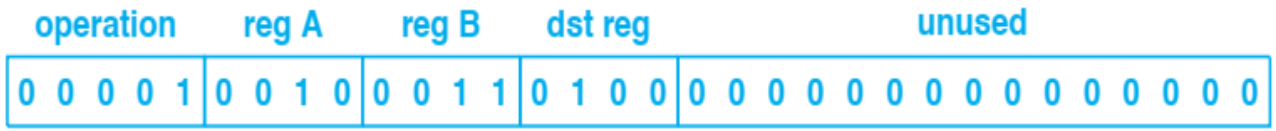
Figure 6.2 The binary representation for each of the four instructions listed in Figure 6.1. Each instruction is thirty-two bits long.

3. Instructions in Memory

- **add r4, r2, r3**

```
add  r4, r2, r3
```

(a)



(b)

Figure 6.3 (a) An example *add* instruction as it appears to a programmer, and (b) the instruction stored in memory.

4. Moving to the Next Instruction

- Programs are stored in memory, and the processor moves through the memory, extracting and executing successive instructions one at a time.
- An *instruction pointer* consists of a register in the processor that holds the memory address of the next instruction to execute.
  - For example, if we imagine a computer with thirty-two-bit memory addresses, an instruction pointer will hold a thirty-two-bit value.
- To execute instructions, the hardware repeats the following three steps.
  - Use the instruction pointer as a memory address and fetch an instruction
  - Use bits in the instruction to control hardware that performs the operation
  - Move the instruction pointer to the next instruction

4. Moving to the Next Instruction

- In our example computer, each instruction occupies thirty-two bits in memory.
  - After an instruction is executed, hardware must increment the instruction pointer by four bytes (thirty two bits) to move to the next instruction.
- Each line in the figure represents a data path that consists of multiple parallel wires. In the figure, each data path is thirty-two bits wide.

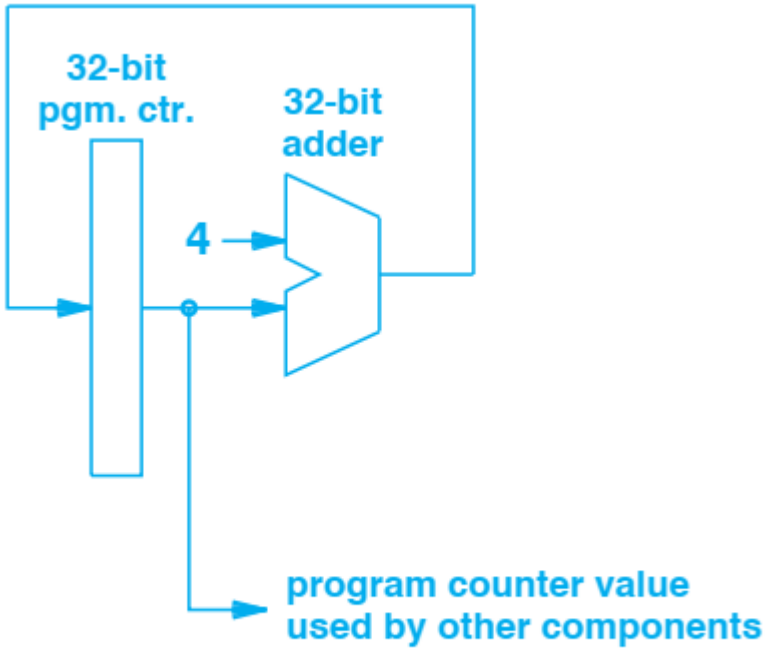


Figure 6.4 Hardware that increments a program counter.

5. Fetching an Instruction

- The next step in constructing a computer consists of fetching an instruction from memory.
- For our simplistic example, we will assume that a dedicated instruction memory holds the program to be executed, and that a memory hardware unit takes an address as input and extracts a thirty-two bit data value from the specified location in memory.

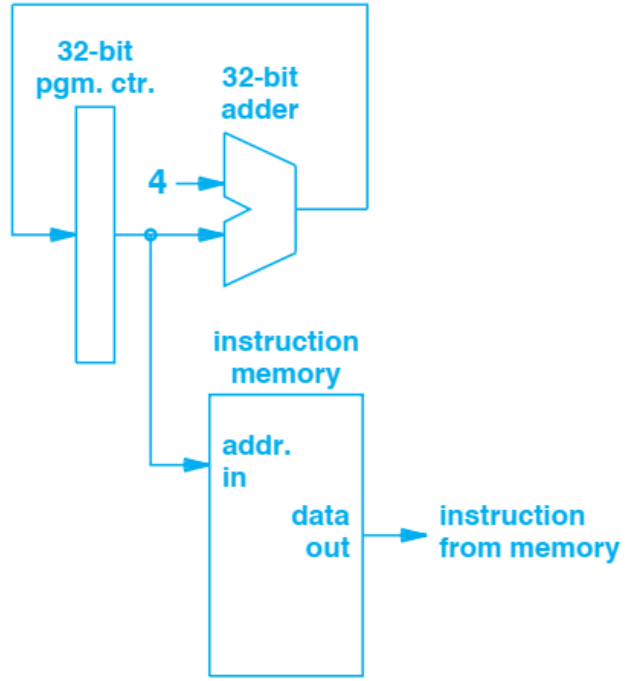


Figure 6.5 The data path used during *instruction fetch* in which the value in a program counter is used as a memory address.

6. Decoding an Instruction

- When an instruction is fetched from memory, it consists of thirty-two bits. The next conceptual step in execution consists of *instruction decoding*. That is, the hardware separates fields of the instruction such as the operation, registers specified, and offset.

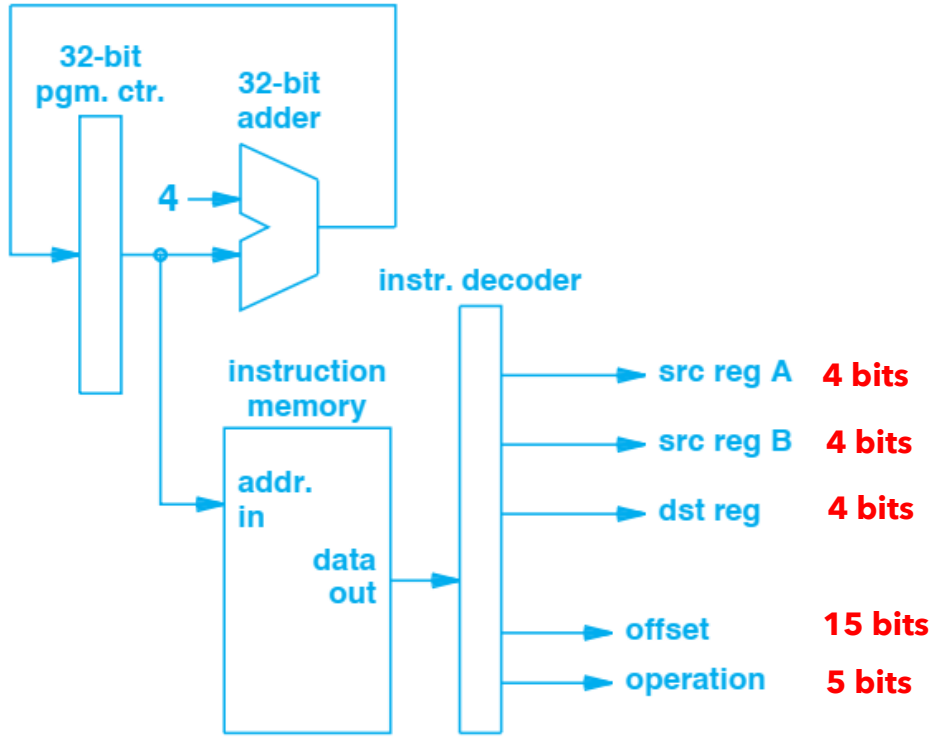


Figure 6.6 Illustration of an instruction decoder connected to the output of the instruction memory.

7. Connections to a Register Unit

- The register fields of an instruction are used to select registers that are used in the instruction.
  - In our example, a *jump* instruction uses one register, a *load* or *store* instruction uses two, and an *add* instruction uses three. Therefore, each of the three possible register fields must be connected to a register storage unit.

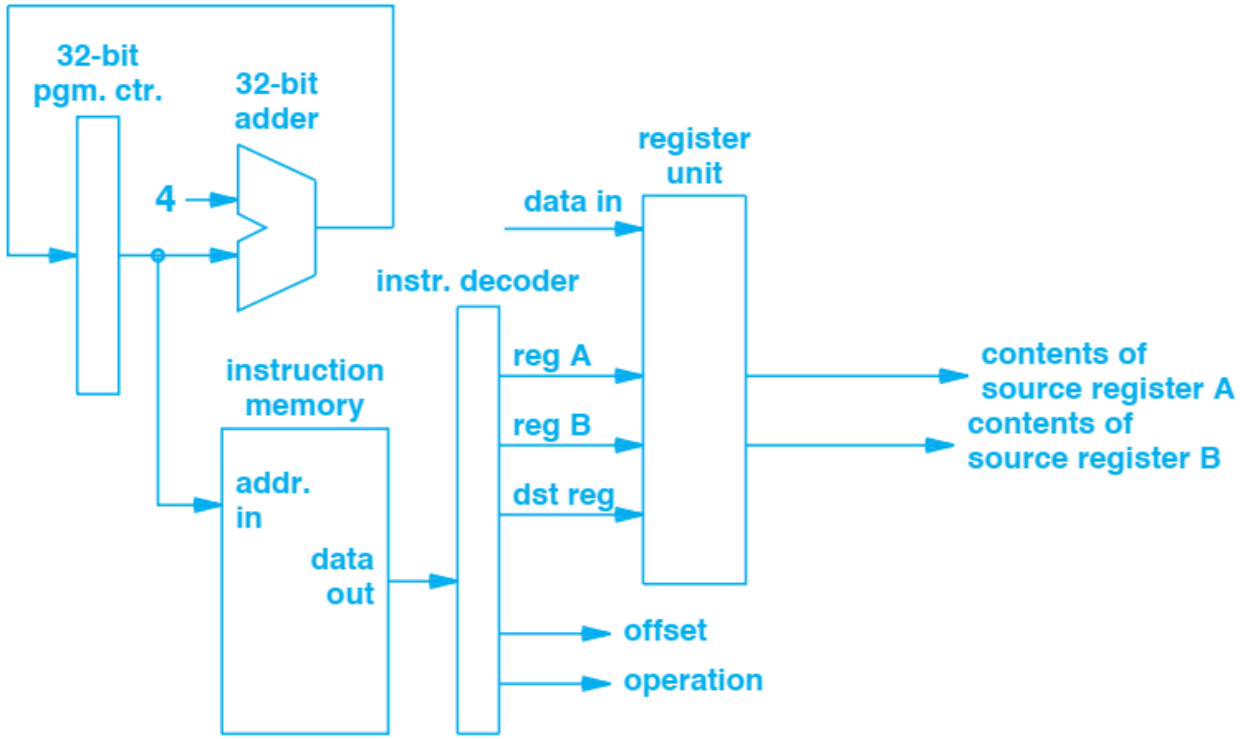


Figure 6.7 Illustration of a register unit attached to an instruction decoder.

8. Control and Coordination

- Although all three register fields connect to the register unit, the unit does not always use all three.
  - Instead, a register unit contains logic that determines whether a given instruction reads existing values from registers or writes data into one of the registers.
  - In particular, the *load* and *add* instructions each write a result to a register, but the *jump* and *store* instructions do not.
  - It may seem that the operation portion of the instruction should be passed to the register unit to allow the unit to know how to act.
    - In practice, most computers use an additional hardware unit, known as a *controller*, to coordinate overall data movement and each of the functional units. A controller must have one or more connections to each of the other units, and must use the *operation* field of an instruction to determine how each unit should operate to perform the instruction.
    - In the diagram, for example, a connection between the controller and register unit would be used to specify whether the register unit should fetch the values of one or two registers, and whether the unit should accept data to be placed in a register. For now, we will assume that a controller exists to coordinate the operation of all units.

9. Arithmetic Operations and Multiplexing

- An important principle: hardware that is designed to re-use functional units.
- Consider arithmetic:
  - The **add** instruction performs arithmetic explicitly. A real processor will have several arithmetic and logical instructions (e.g., **subtract**, **shift**, **logical** and, etc), and will use the **operation** field in the instruction to decide which the ALU should perform.
  - Our instruction set also has an implicit arithmetic operation associated with the **load**, **store**, and **jump** instructions. Each of those instructions requires an addition operation to be performed when the instruction is executed.
    - Namely, the processor must add the offset value, which is found in the instruction itself, to the contents of a register. The resulting sum is then treated as a memory address.
- The question arises: should a processor have a separate hardware unit to compute the sum needed for an address, or should a single ALU be used for both general arithmetic and address arithmetic?
  - Separate functional units have the advantage of speed and ease of design. Re-using a functional unit for multiple purposes has the advantage of taking less power

9. Arithmetic Operations and Multiplexing

- Our design illustrates re-use. A single *Arithmetic Logic Unit (ALU)* performs all arithmetic operations.
- Inputs to the ALU can come from two sources:
  - A pair of registers
  - A register and the offset field in an instruction.
- How can a hardware unit choose among multiple sources of input? The mechanism that accommodates two possible inputs is known as a *multiplexor*.

9. Arithmetic Operations and Multiplexing

- In the figure, inputs to the multiplexor come from the register unit and the offset field in the instruction.
- How does the multiplexor decide which input to pass along?
  - Recall that our diagram only shows the data path. In addition, the processor contains a controller, and all units are connected to the controller. When the processor executes an **add** instruction, the controller signals the multiplexor to select the input coming from the register unit. When the processor executes other instructions, the controller specifies that the multiplexor should select the input that comes from the offset field in the instruction.

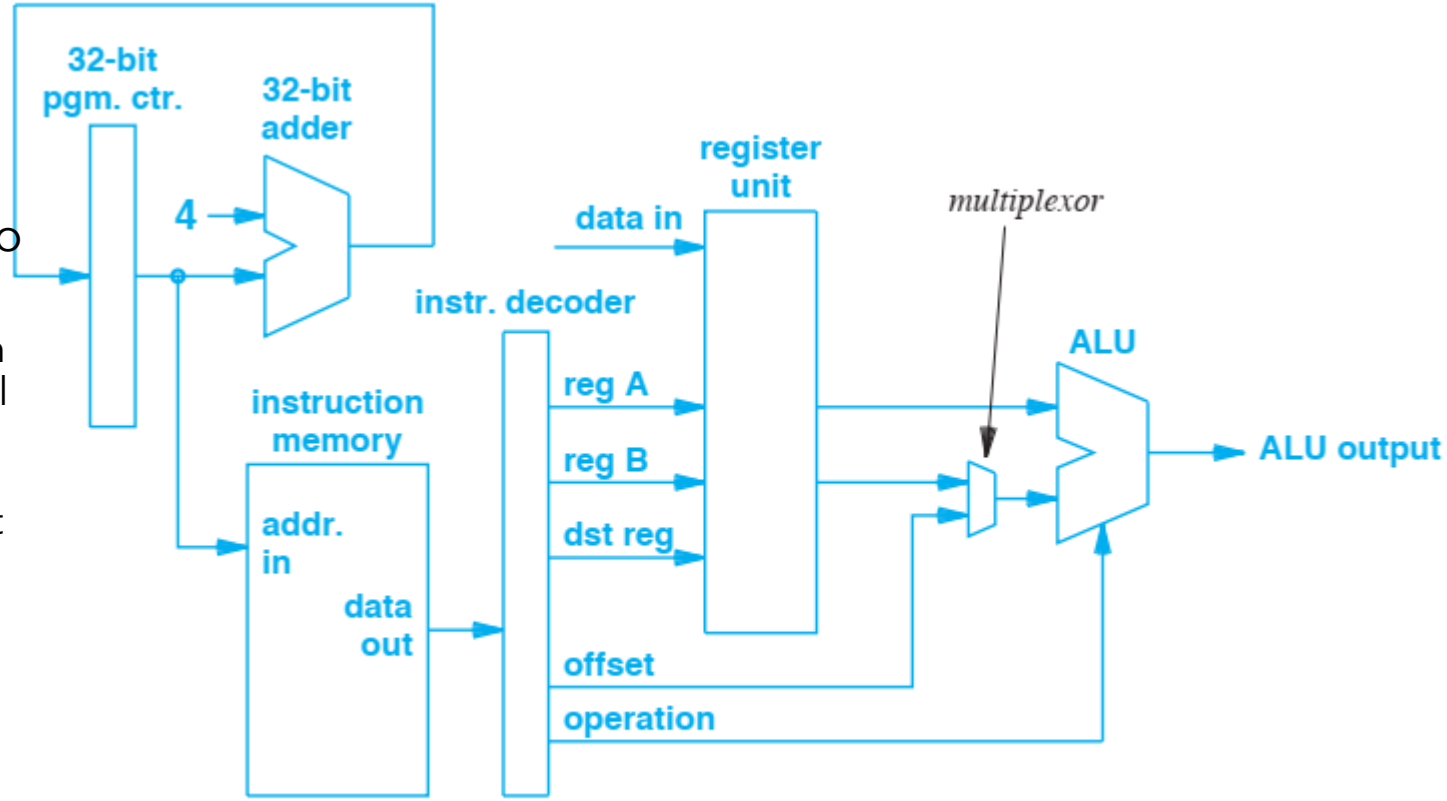


Figure 6.8 Illustration of a multiplexor used to select an input for the ALU.

10. Operations Involving Data in Memory

- When it executes a **load** or **store** operation, the computer must reference an item in the data memory. For such operations, the ALU is used to add the offset in the instruction to the contents of a register, and the result is used as a memory address.
- In our simplified design, the memory used to store data is separate from the memory used to store instructions.

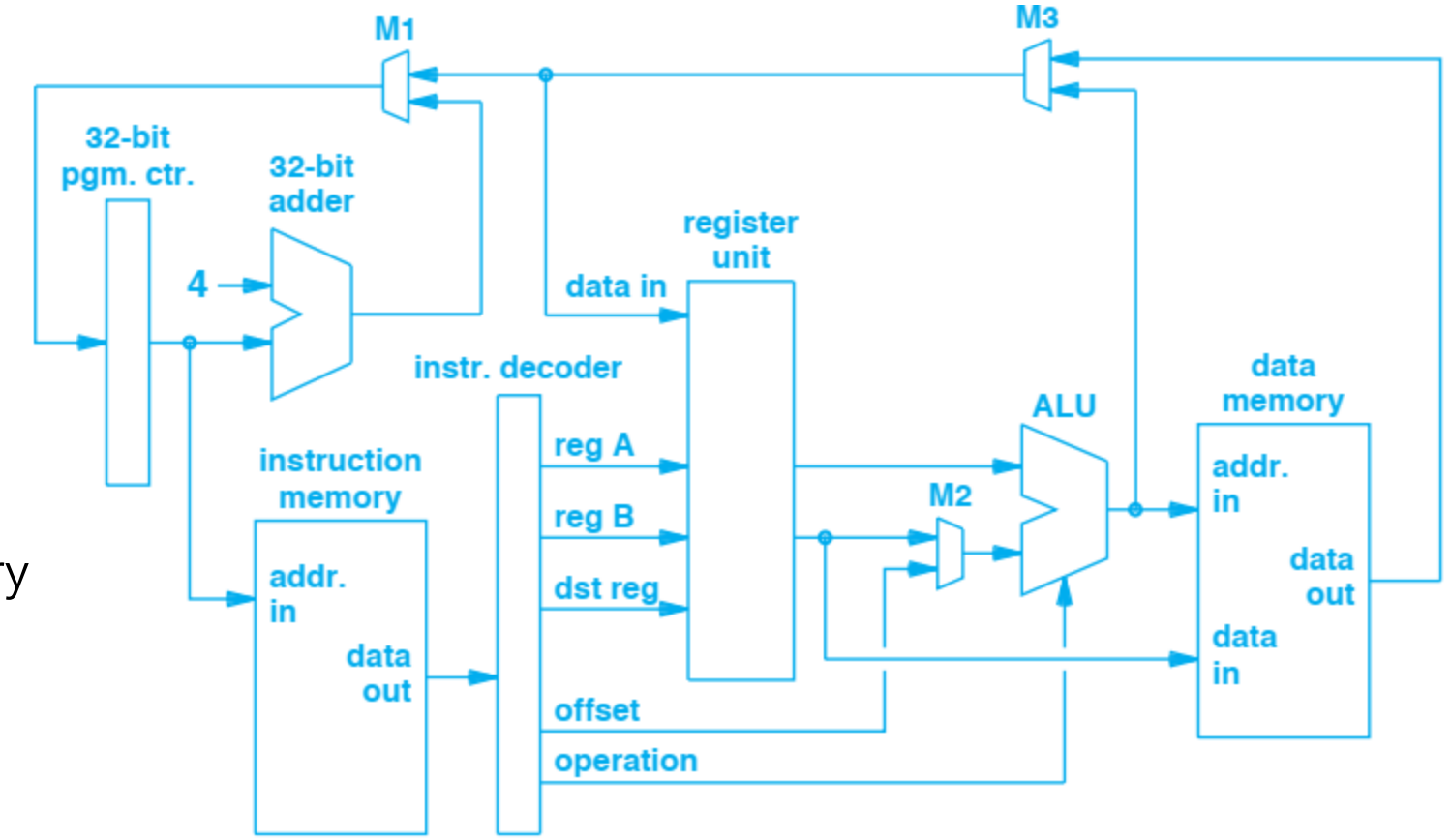


Figure 6.9 Illustration of data paths including data memory.