

# Part 1

## CHAPTER 3

### Architecture and Organization



1

These slides are provided with permission from the copyright for CS2208 use only. The slides must not be reproduced or provided to anyone outside the class.

All downloaded copies of the slides are for personal use only.

Students must destroy these copies within 30 days after receiving the course's final assessment.

# The Instruction Set Architecture

In this chapter, we will:

- ❑ **Revisit** the *stored program machine* and **show** how an instruction is executed
- ❑ **Introduce** instruction formats, including
  - *memory-to-register*,
  - *register-to-memory*, and
  - *register-to-register*
- ❑ **Demonstrate** how a processor implements *conditional behavior*
- ❑ **Describe** a set of computer assembly instructions (*instructions set*)
- ❑ **Show** how computers access data (*addressing modes*)
- ❑ **Introduce** an ARM's *Integrated Development Environment* (IDE) and **show** how ARM programs are written

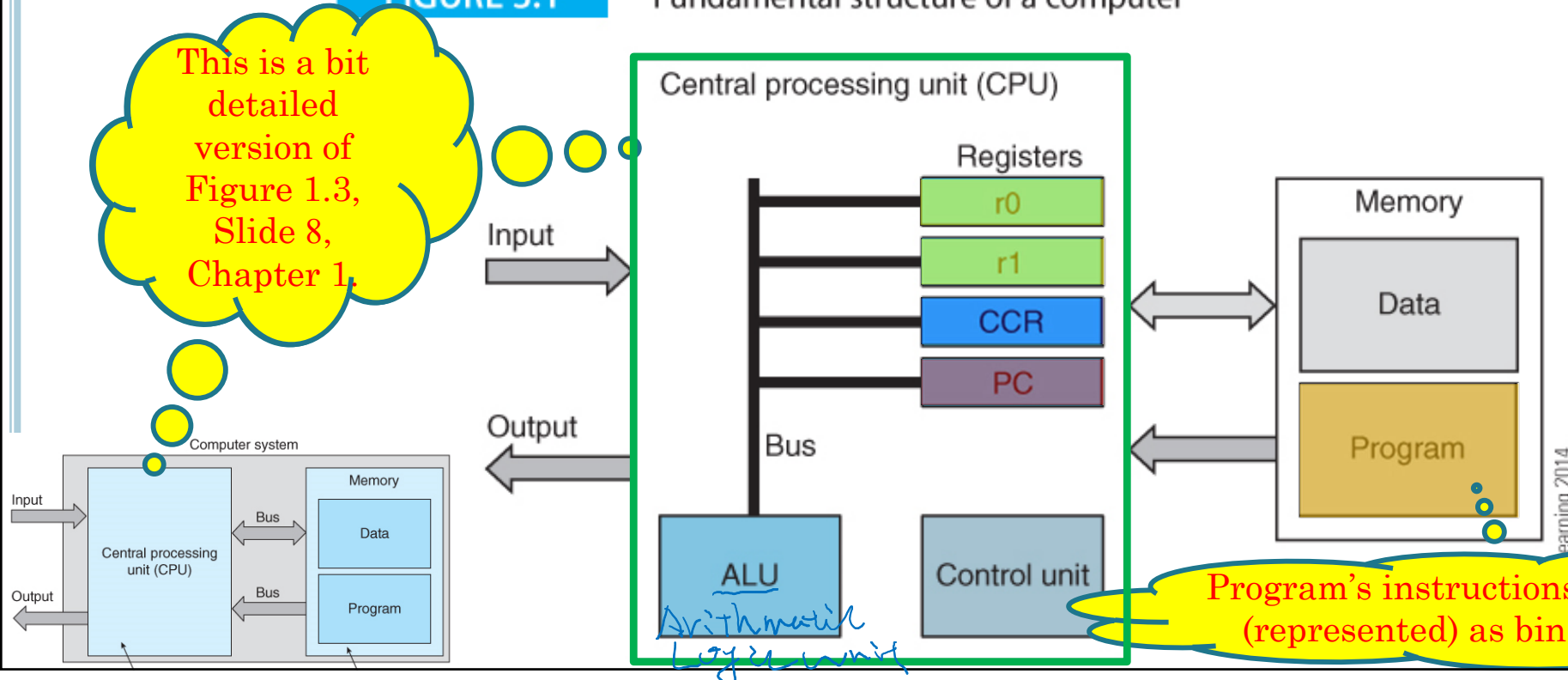
# The Instruction Set Architecture

- ❑ Figure 3.1 illustrate the structure of a **simple *hypothetical stored program computer***.
- ❑ The **CPU** reads instructions from memory and executes them.
- ❑ *Temporary data* is stored in **registers** such as *r0* and *r1*.
- ❑ The **PC**, *program counter*, is the register that points at (i.e., contains the address of) the next instruction to be executed.
- ❑ The **CCR**, *Condition Code Register*, is a collection of flag bits for a processor.

Intel calls it the "**Instruction Address Pointer**", a better name than the **PC**

FIGURE 3.1

Fundamental structure of a computer



## Instruction Formats

- ❑ A computer executes instructions from 8-bits wide to multi-bytes wide.
- ❑ The **instruction format** defines the *anatomy of an instruction*
  - the **number of operands**, and
  - the number of **bits** devoted to **defining each operation**,
  - the **format of each operand**.
- ❑ Below are several *hypothetical* examples of assembly instructions:

**LDR** **registerDestination**,memorySource

**STR** registerSource,**memoryDestination**

**Operation** **registerDestination**,registerSource1,registerSource2

LDR     **r1**, 1234

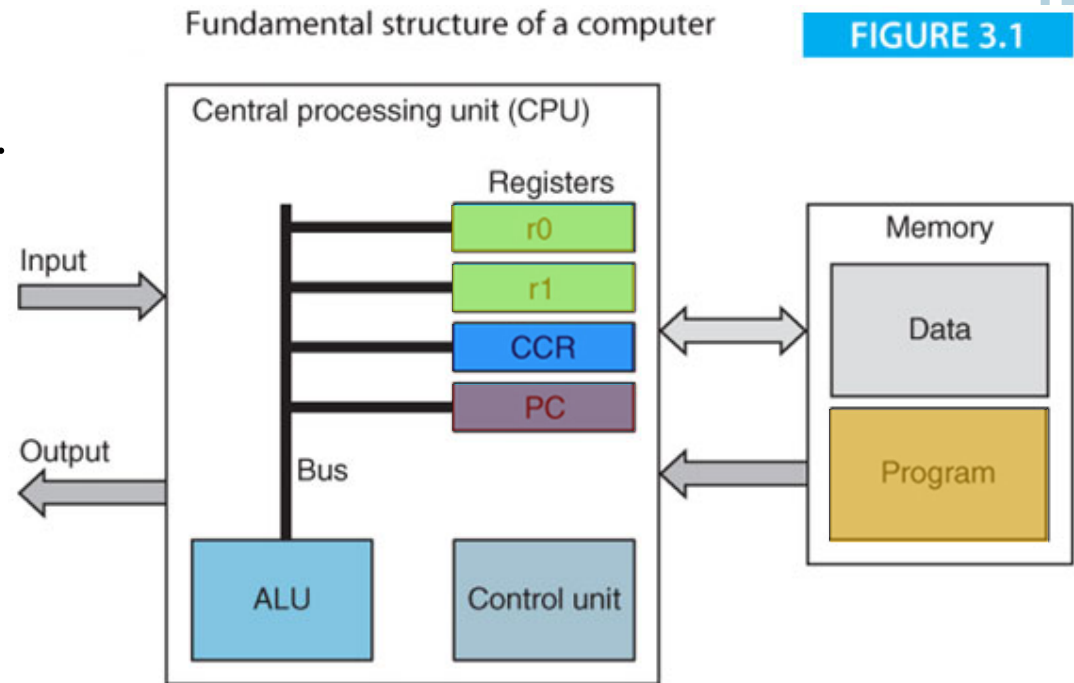
STR     r3, **2000**

ADD     **r1**, r2, r3

SUB     **r3**, r3, r1

# Features

- ❑ A stored program machine is a computer that has a program in binary form in its main memory.
- ❑ The program and data are stored in the same memory.
- ❑ The program counter (PC) points to the next instruction to be executed and is incremented after executing each instruction.
- ❑ A stored program operates in a *fetch/execute two-phase mode*.
  - In the *fetch phase* the next instruction is read from memory and decoded.
  - In the *execute phase* the instruction is interpreted and executed by the CPU's logic.
- ❑ Modern computers are *pipelined*, where *fetch* and *execute* operations overlap.



Review Slides 24 and 25 in Chapter 1.

# Features

A stored program computer has several registers.

**r0 - ri** The register file is a *set of general-purpose registers*, e.g., r0, r1, r2, ..., ri that store temporary (working) data, for example, the intermediate results of calculations, where *i* is typically 8, 16, or 32.

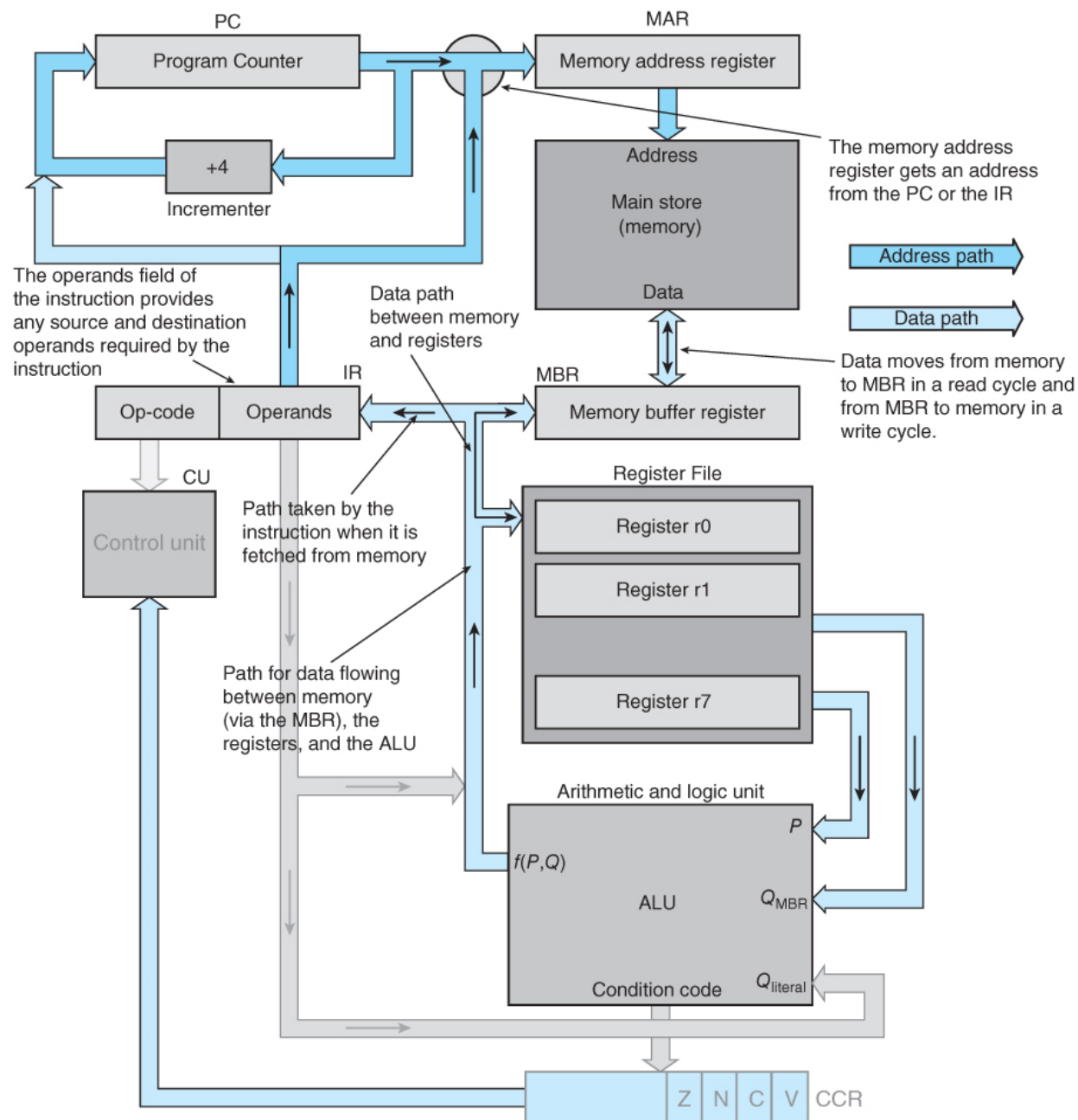
A computer requires at *least one* general-purpose register.

**PC** The *program counter* contains the *address* of the next instruction to be executed. Thus, the PC *points* to the location in memory that holds the next instruction.

**IR** The *instruction register* stores the instruction most recently read from main memory. This is the instruction currently being executed.

**MAR** The *memory address register* stores the *address* of the location in main memory that is currently being accessed by a *read* or *write* operation.

**MBR** The *memory buffer register* stores *data* that has just been *read* from main memory, or *data* to be immediately *written* to main memory.

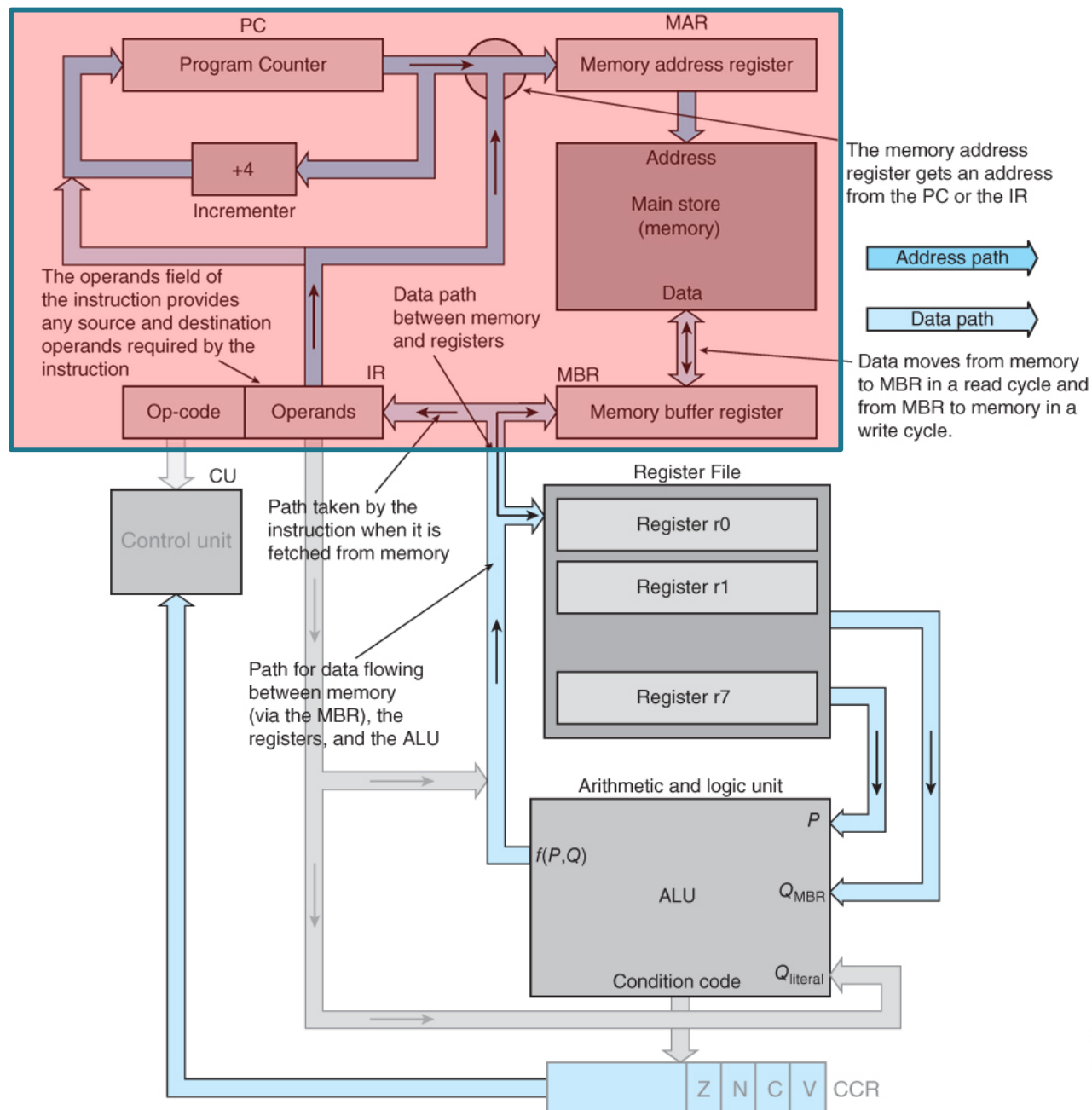
**FIGURE 3.2** Partial structure of a hypothetical stored program machine

## Structure of a Computer

□ We are going to use an ARM processor to introduce assembly language and a modern ISA.

□ *However*, it would be better to begin with the description of a very simple hypothetical computer to keep things simple.

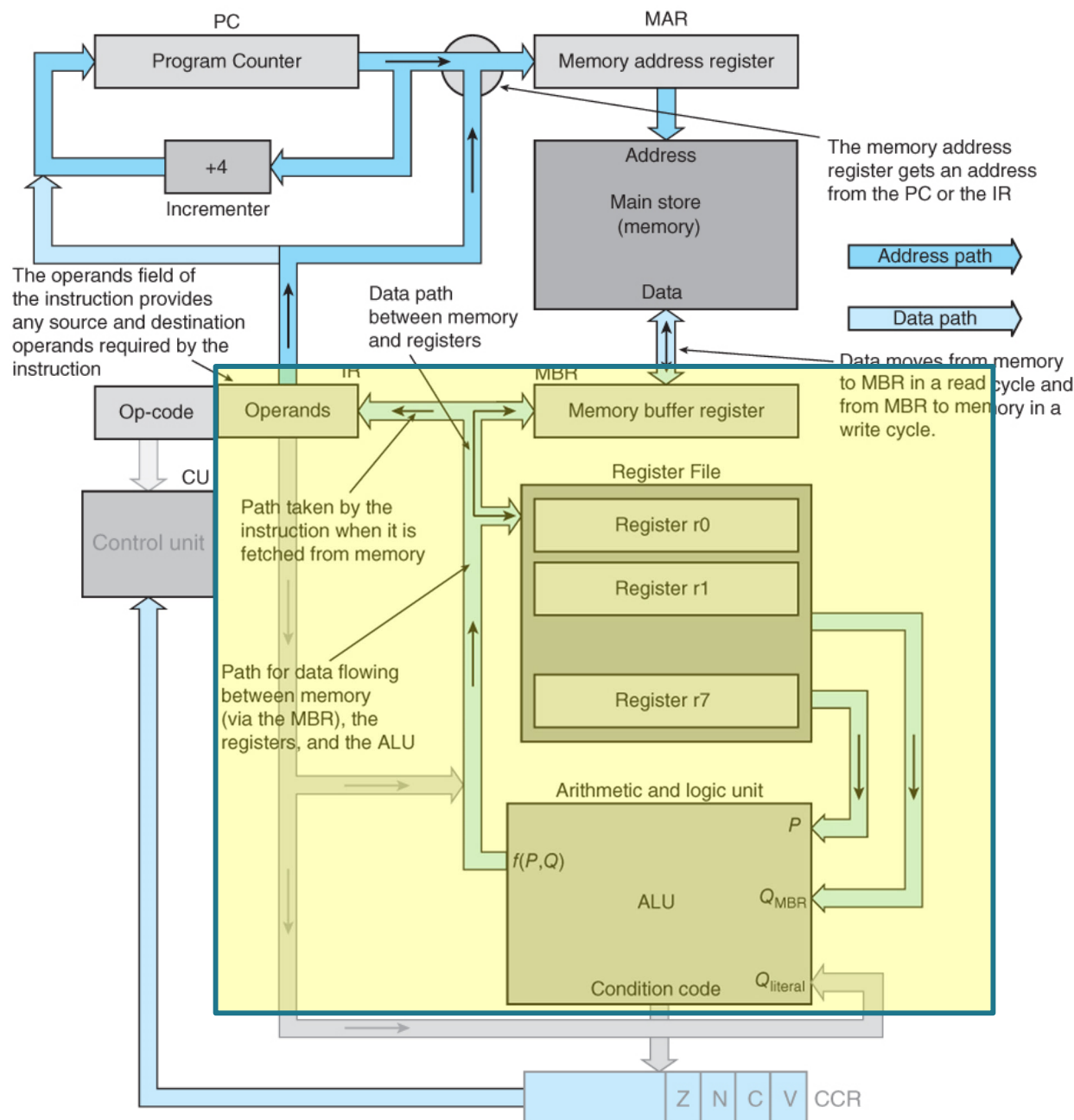


**FIGURE 3.2** Partial structure of a hypothetical stored program machine

## Structure of a Computer

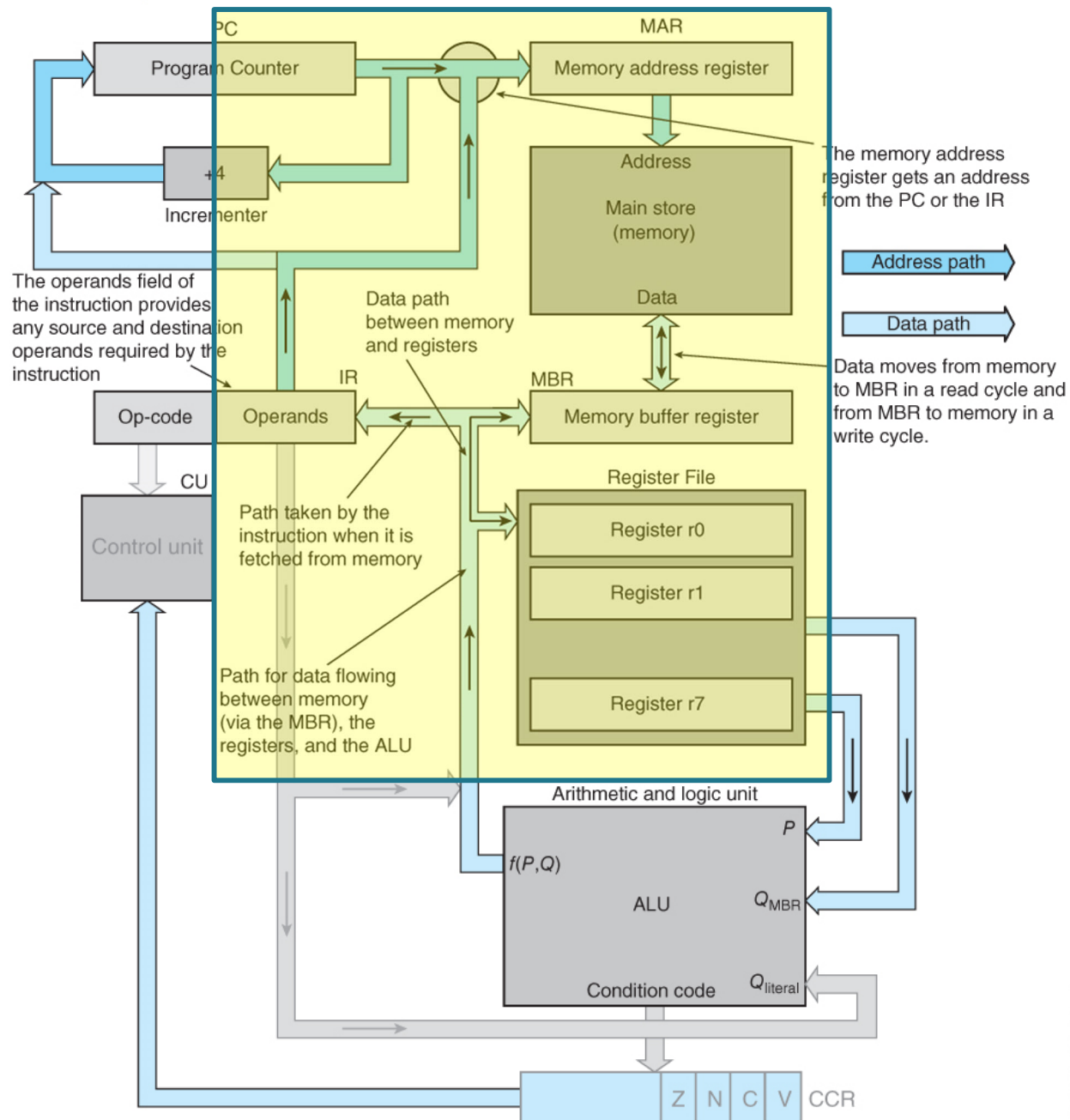
- ❑ In the *fetch phase*, the Program Counter, **PC**, supplies the address of the next instruction to be executed to the **MAR** to read this instruction and the PC is *incremented by the size of an instruction*.
- ❑ The instruction is read and loaded into the Memory Buffer Register, **MBR**, and then copied to the Instruction Register, **IR**, where the op-code is decoded.



**FIGURE 3.2** Partial structure of a hypothetical stored program machine

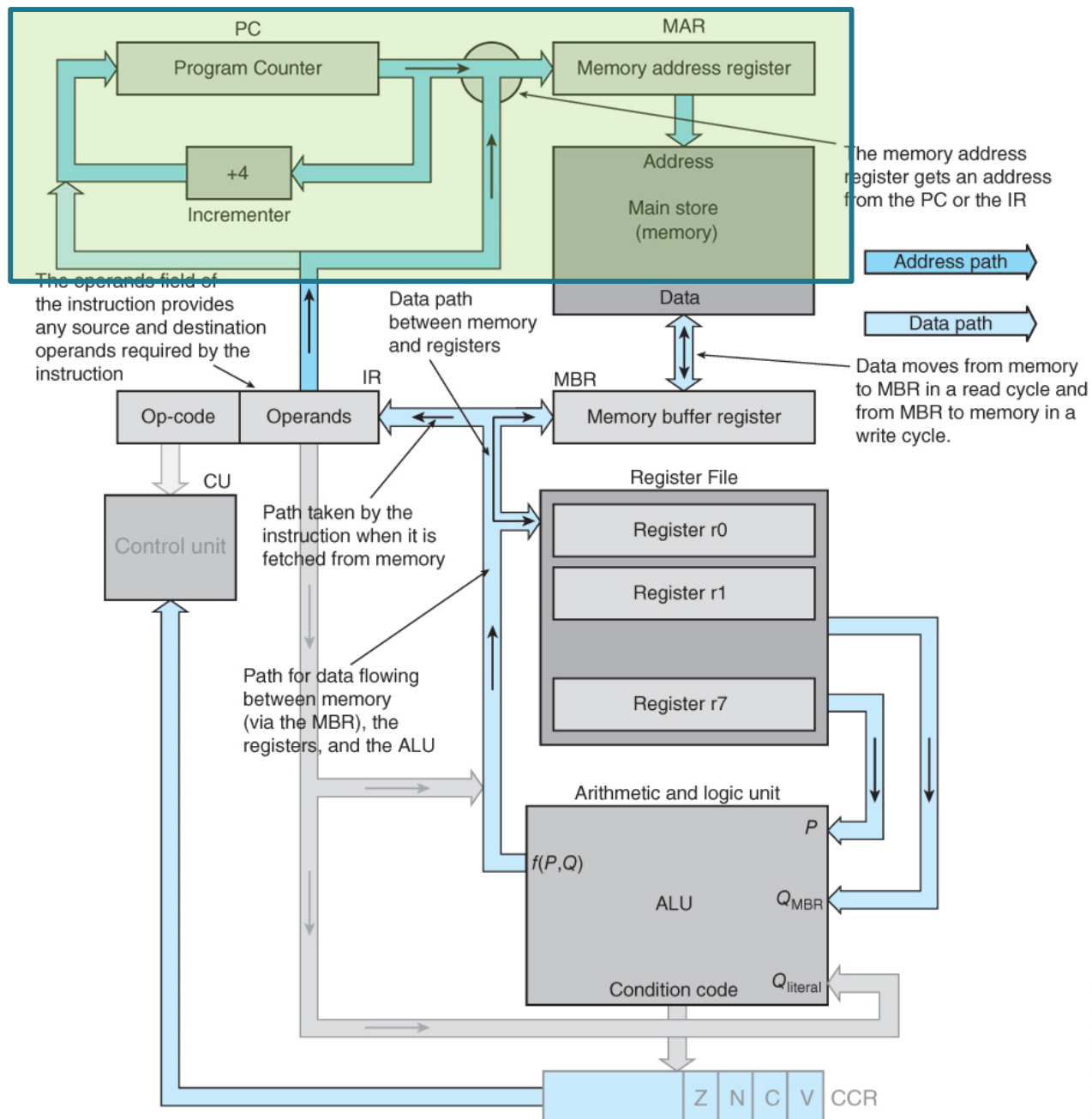
## Structure of a Computer

- In the *execute phase*, the operands may be read from the *register file*, transferred to the *ALU (arithmetic and logic unit)* where they are operated on and then the result passed to the *destination register*. This is what we call, *register-to-register* operation.

**FIGURE 3.2** Partial structure of a hypothetical stored program machine

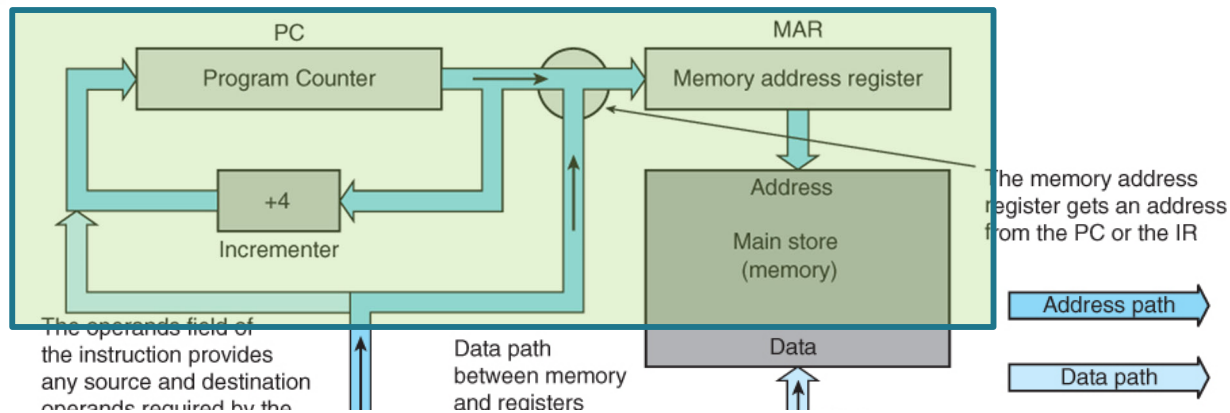
## Structure of a Computer

- If the operation requires a memory access (e.g., a load or store), the memory address in the instruction register is sent to the **MAR** and a read or write operation performed.

**FIGURE 3.2** Partial structure of a hypothetical stored program machine

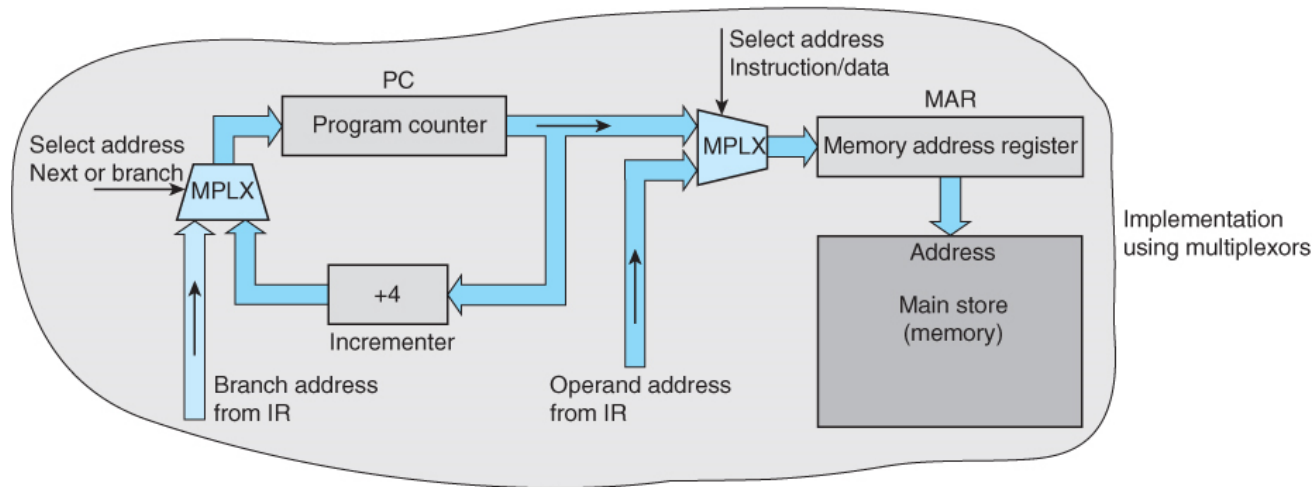
## Structure of a Computer

- But how can we combine two input data lines together?

**FIGURE 3.2** Partial structure of a hypothetical stored program machine

## Structure of a Computer

- But how can we combine two input data lines together?



© Cengage Learning 2014

# Structure of a Computer

## □ Fetch/execute cycle in RTL .

Review Slide 28 in Chapter 1.

**FETCH**  $[MAR] \leftarrow [PC]$  ; Step 1: copy PC to MAR  
 $[PC] \leftarrow [PC] + 4$  ; Step 1: increment PC

$[MBR] \leftarrow [[MAR]]$  ; Step 2: read instruction pointed at by MAR  
 $[IR] \leftarrow [MBR]$  ; Step 3: copy instruction in MBR to IR

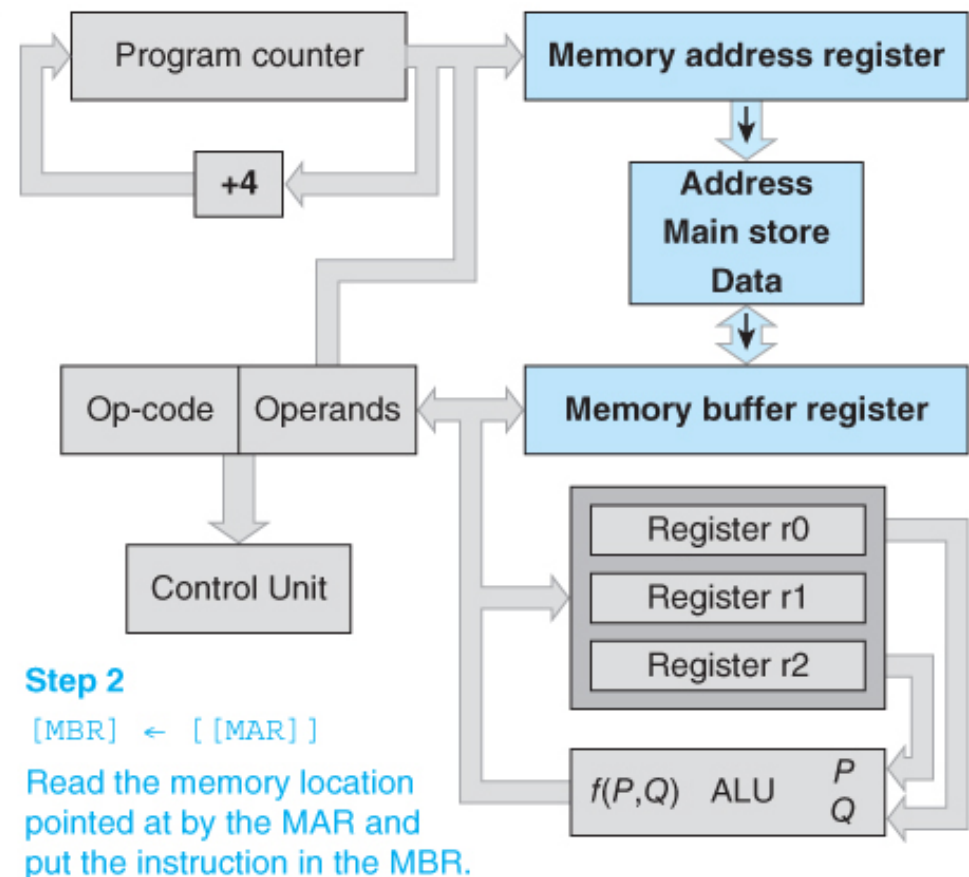
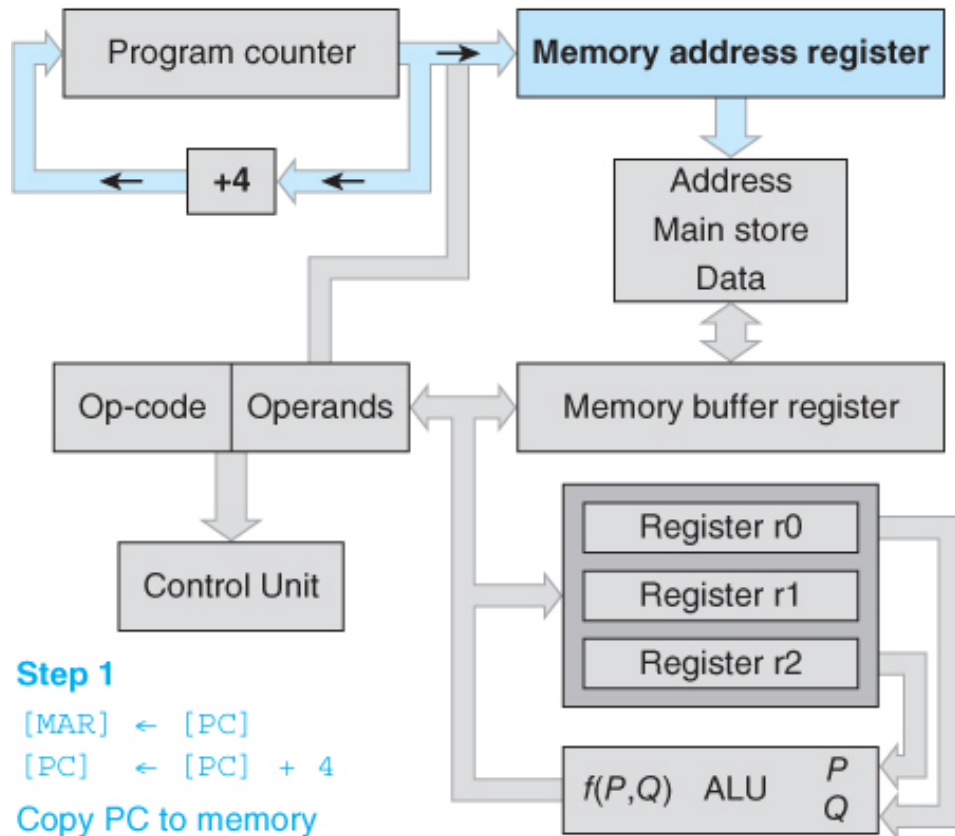
## EXECUTE

**LDR**  $[MAR] \leftarrow [IR(\text{address})]$  ; Step 4: copy operand address from IR to MAR  
 $[MBR] \leftarrow [[MAR]]$  ; Step 5: read operand value from memory  
 $[r1] \leftarrow [MBR]$  ; Step 6: copy the operand to a register, e.g., r1

Review Slides 24 and 25 in Chapter 1.

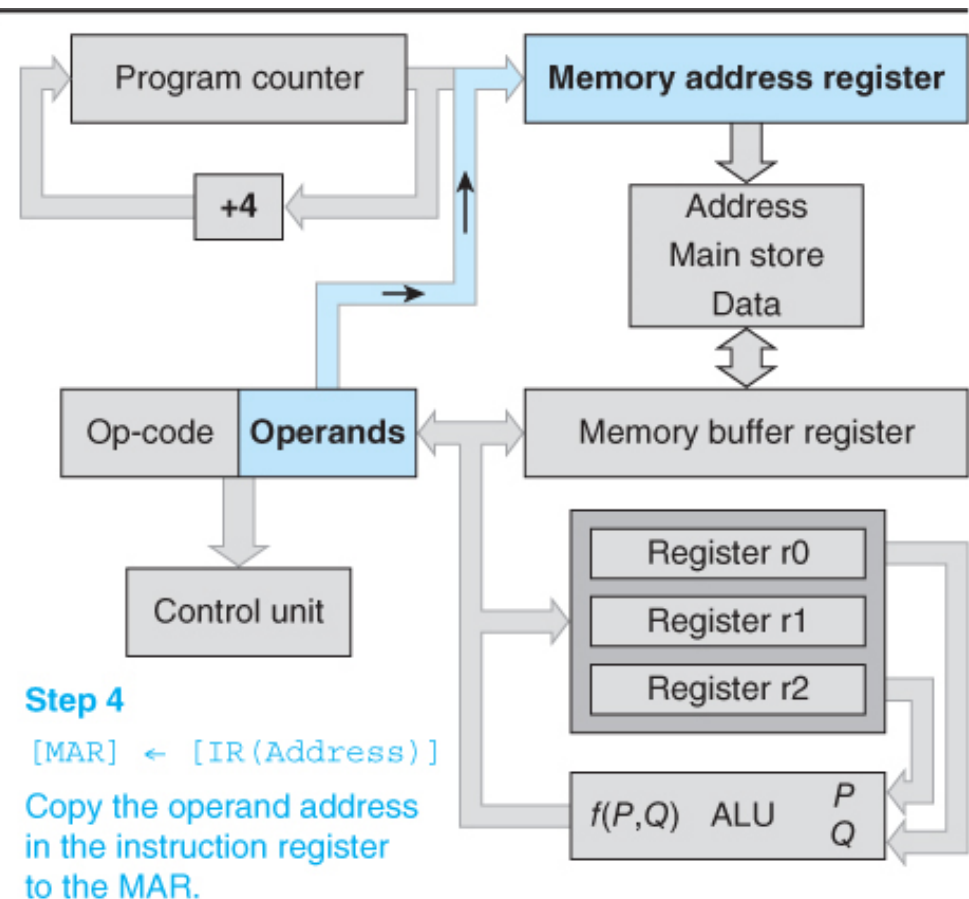
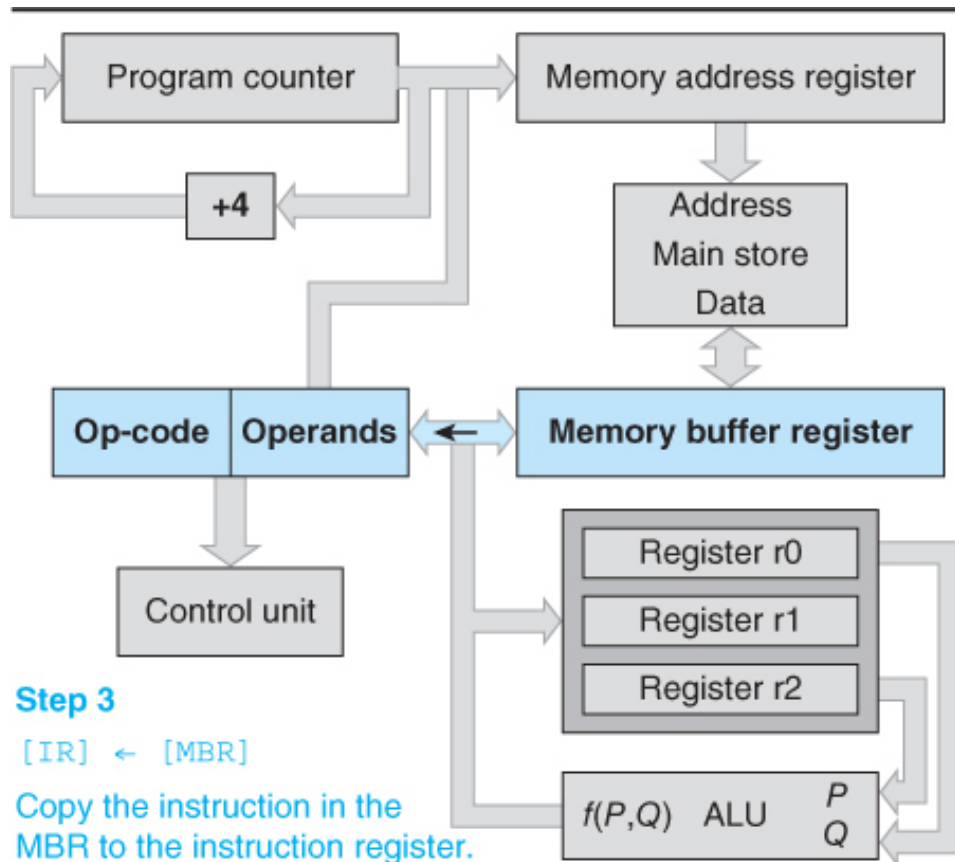
The coming 3 slides show the above steps graphically.

# Fetching and Executing an Instruction

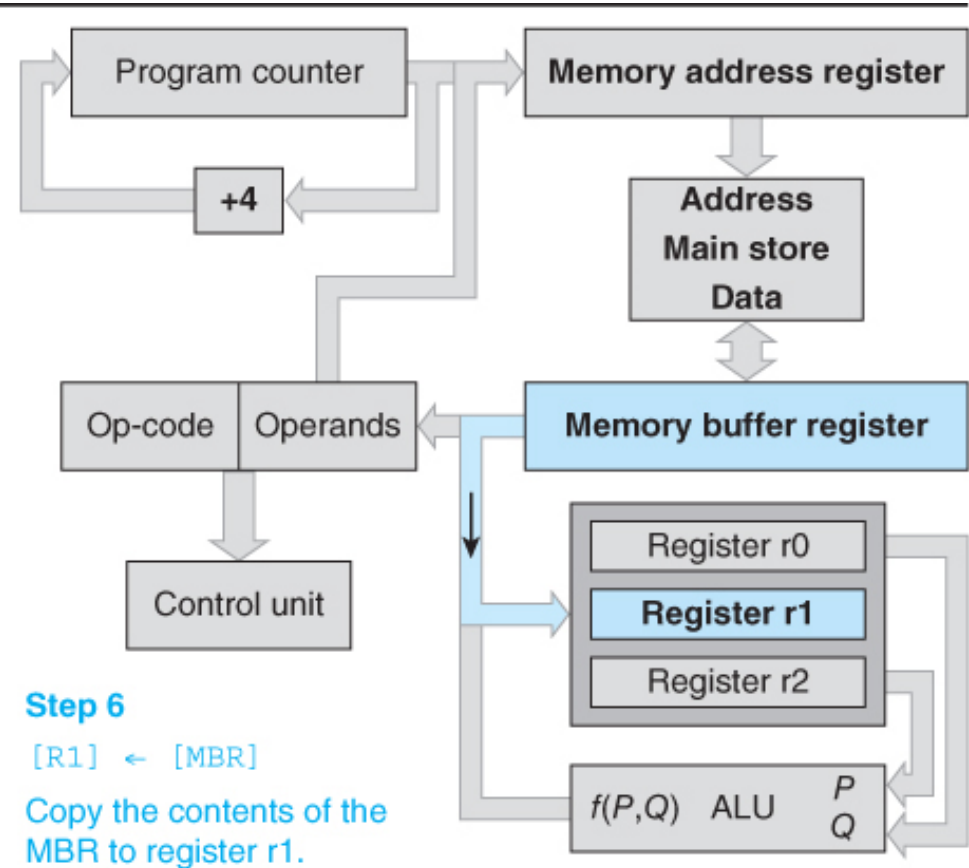
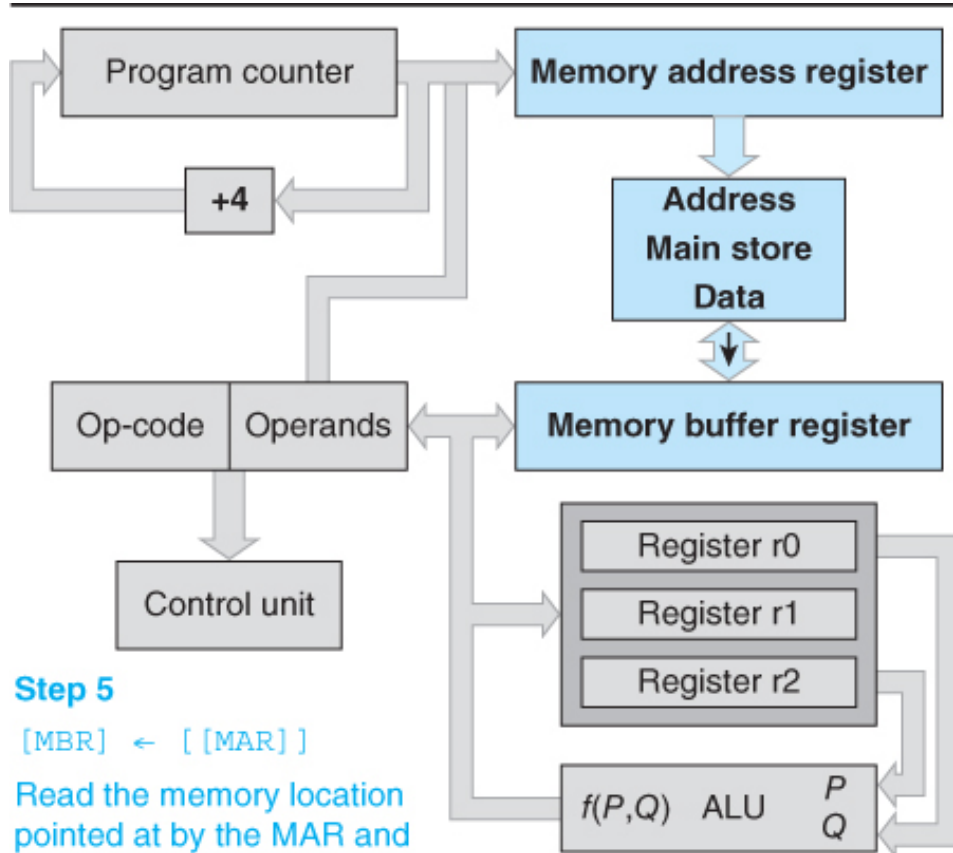




# Fetching and Executing an Instruction

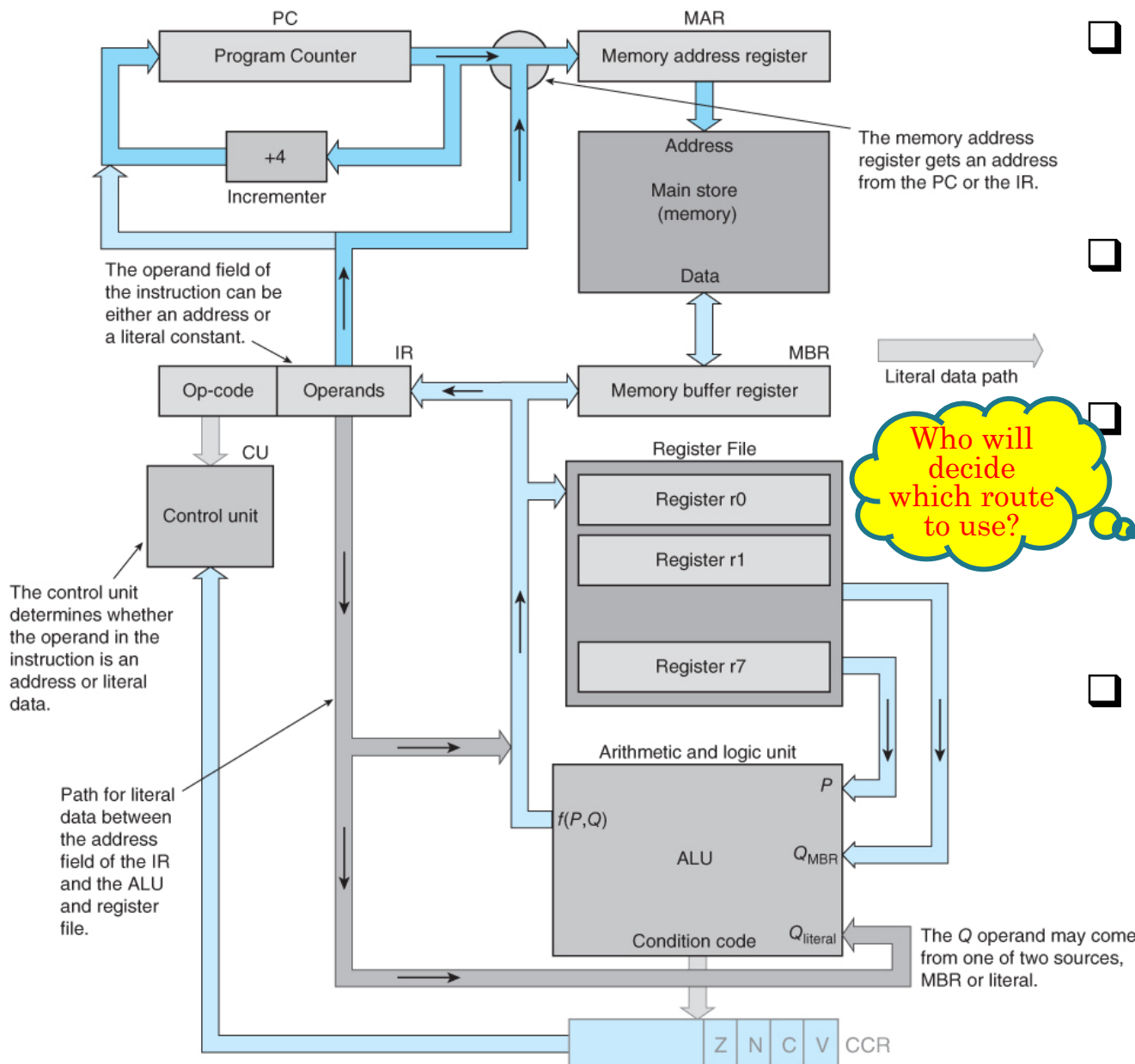


# Fetching and Executing an Instruction



## Dealing with Constants

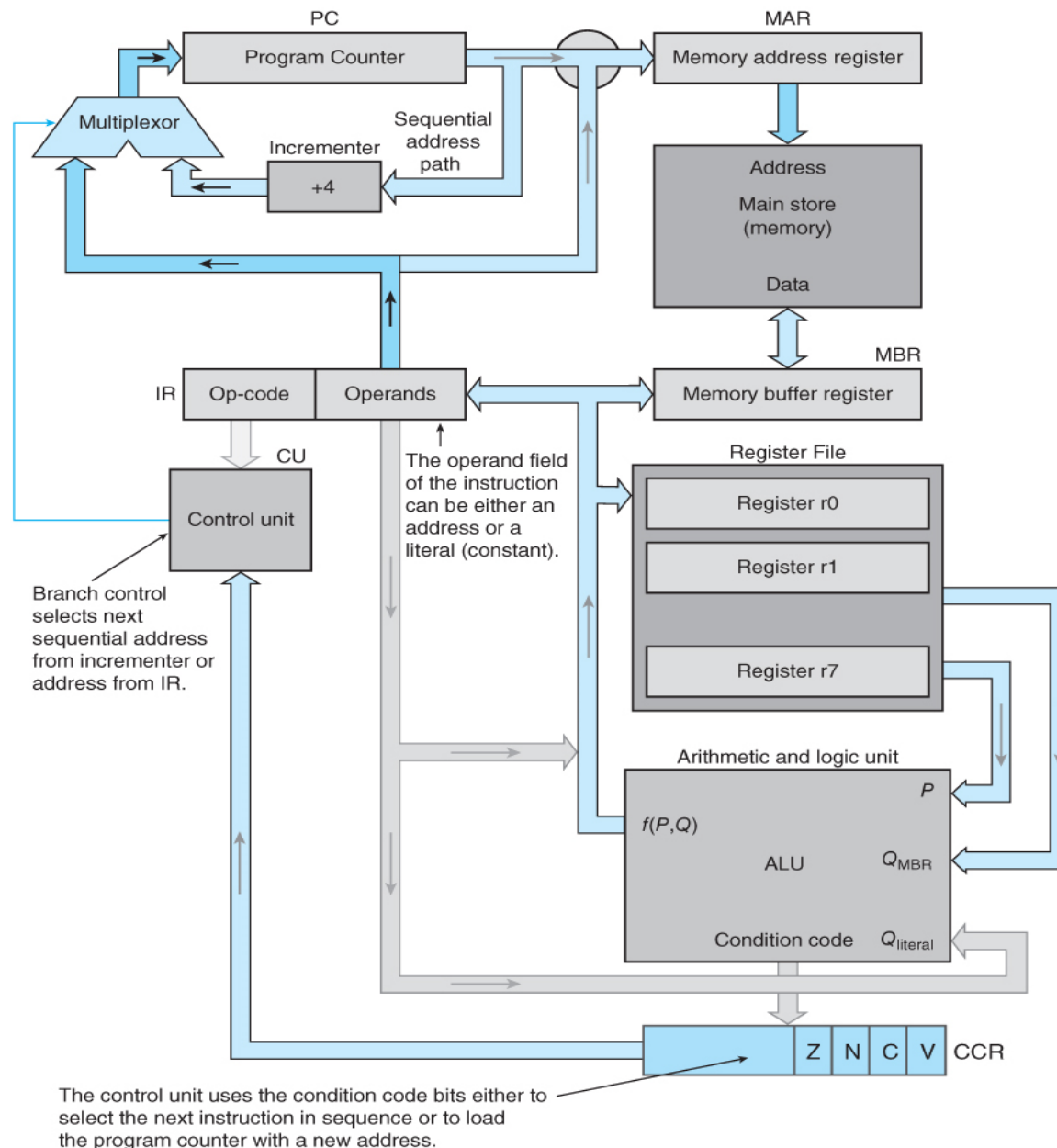
FIGURE 3.4 Information paths for literal operands



- ❑ Suppose we want to **load the number 1234 itself (a.k.a. literal operand)** into register r1.
- ❑ **ADD r0,r1,#25** adds the value 25 to the content of r1 and puts the sum in r0
- ❑ A path from the instruction register, **IR**, routes a literal operand to **either the register file, MBR, and ALU**
- ❑ When **ADD r0,r1,#25** is executed, the operand to be added to r1, i.e., **#25**, is routed from the operand field of the **IR**, rather than from registers.

## Flow Control

FIGURE 3.5 Implementing conditional behavior at the machine level

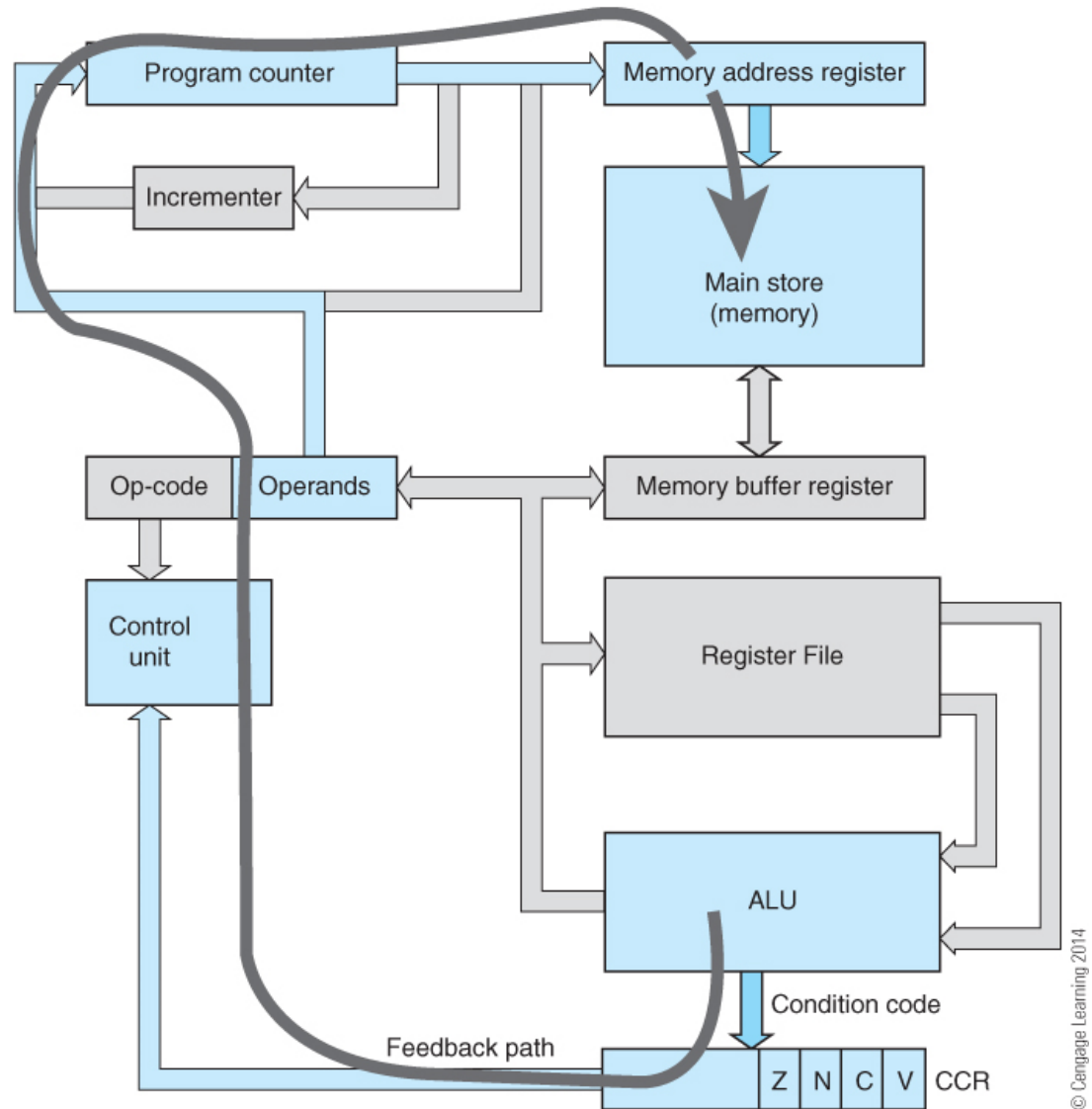


- ❑ *Flow control* refers to any action that *modifies* the normal instruction sequence.
- ❑ *Conditional behavior* allows a processor (*based on the values in the CCR register*) to select one of two possible courses of actions:
  - Continuing executing the *next instruction* in sequence, or
  - Loading the Program Counter with a new value and executing a *branch to another region* of code.

# Flow Control

**FIGURE 3.6** Feedback from ALU to instruction


Figure 3.6 illustrate how the result from the ALU can be used to modify the sequence of instructions.



© Cengage Learning 2014

## Status Bits (Flags)

- ❑ When a computer performs an operation, it stores the *status* or *condition* information in the *Condition Code Register (CCR)*.
- ❑ The processor records whether the result is
  - **Zero (Z)**,
  - **Negative in two's complement terms (N)**,
  - **generated a Carry (C)**, or • • •
  - **generated an arithmetic oVerflow (V)**.



This is the  
carry-out



## Status Bits (Flags)

- Example (*assume that we are dealing with an 8-bit processor*):

00110011	11111111	01011100	11011100
+01000010	+00000001	+01000001	+11000001
<u>01110101</u>	<u>10000000</u>	<u>10011101</u>	<u>11001101</u>
Z = 0, N = 0	Z = 1, N = 0	Z = 0, N = 1	Z = 0, N = 1
C = 0, V = 0	C = 1, V = 0	C = 0, V = 1	C = 1, V = 0
51	-1	92	-36
+66	+1	+65	-63
---	---	---	---
117	0	-99	-99

**CISC means COMPLEX Instruction Set Computer**

- **CISC** processors, like the *Intel IA32*,
- automatically update status flags after each operation.

**RISC means REDUCED Instruction Set Computer**

- **RISC** processors, like the *ARM*,
- require the programmer to request updating the status flags.
- In *ARM* processors, *programmers need to request updating the status flags by appending an S to the instruction*;
- for example, SUBS (instead of SUB) or ADDS (instead of ADD).