

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

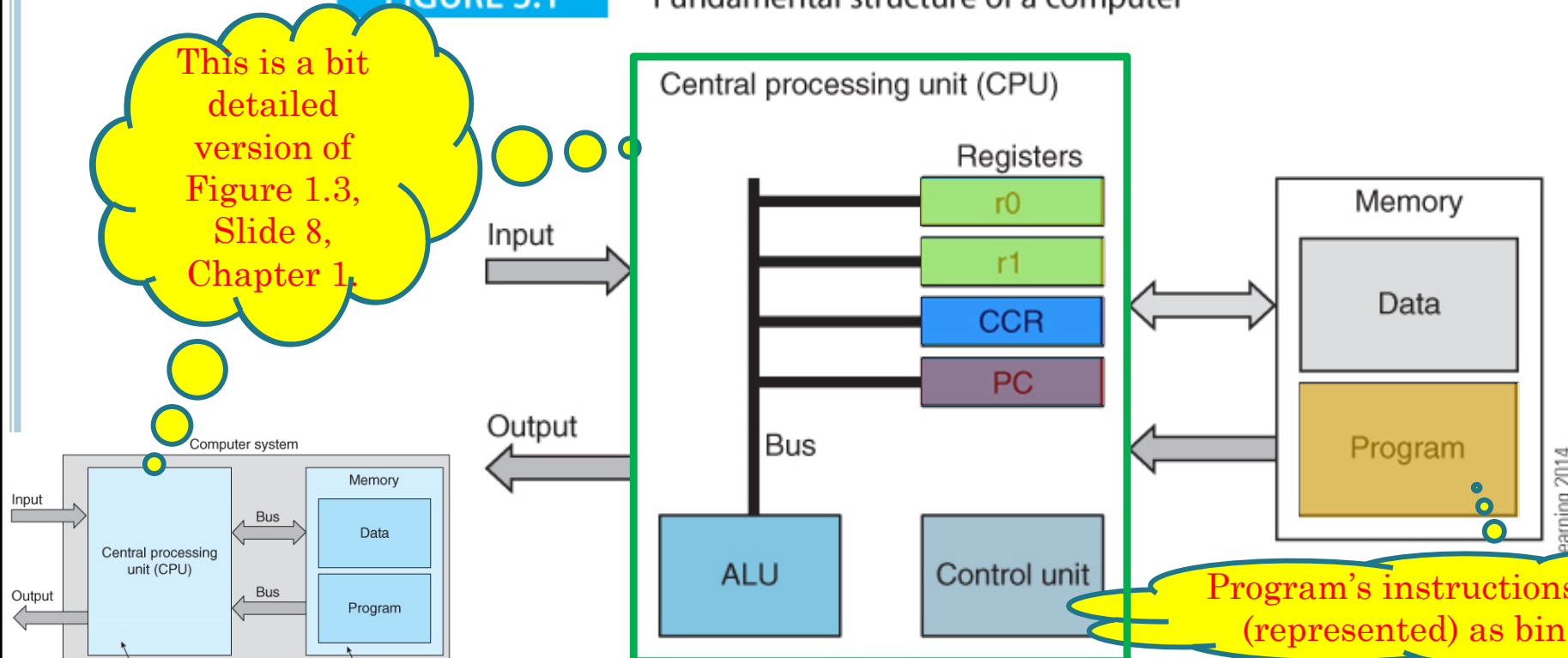
The Instruction Set Architecture

- ❑ 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



This is a bit detailed version of Figure 1.3, Slide 8, Chapter 1.

Program's instructions are encoded (represented) as binary values.

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

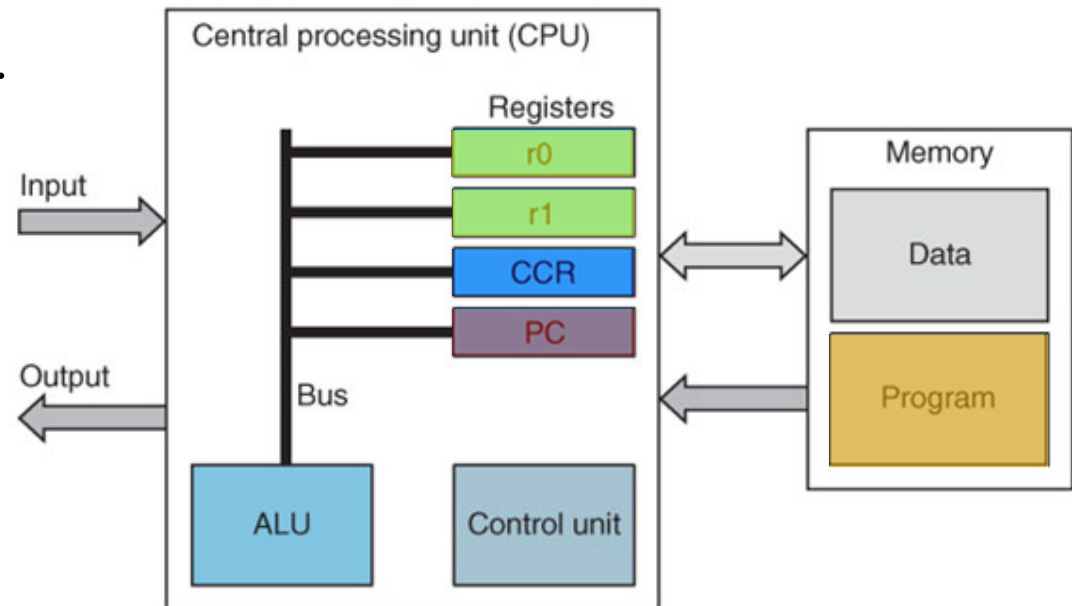
SUB **r3**, r3, r1

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*.

Fundamental structure of a computer

FIGURE 3.1

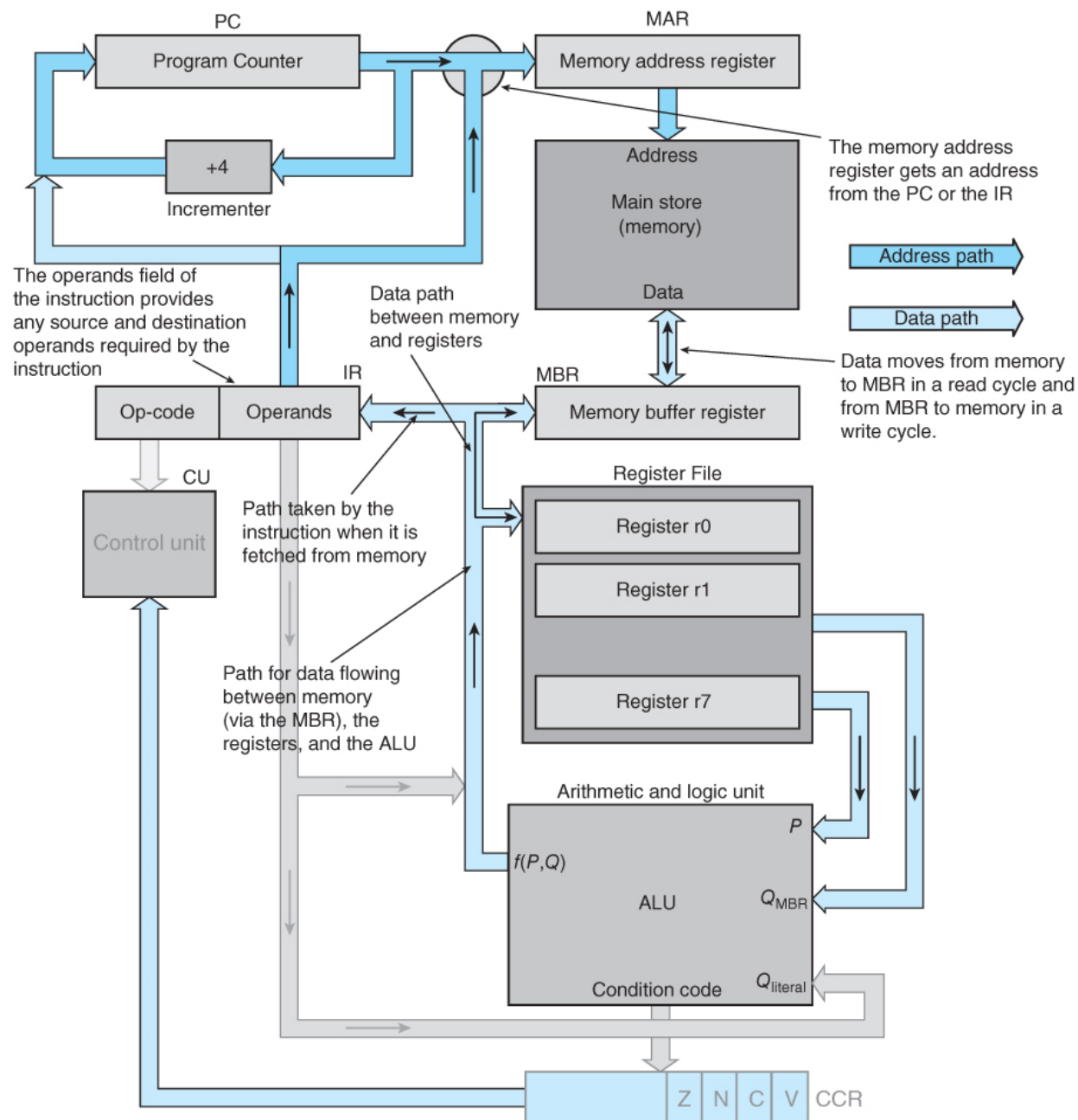


Review Slides 24 and 25 in Chapter 1.

Features

A stored program computer has several registers.

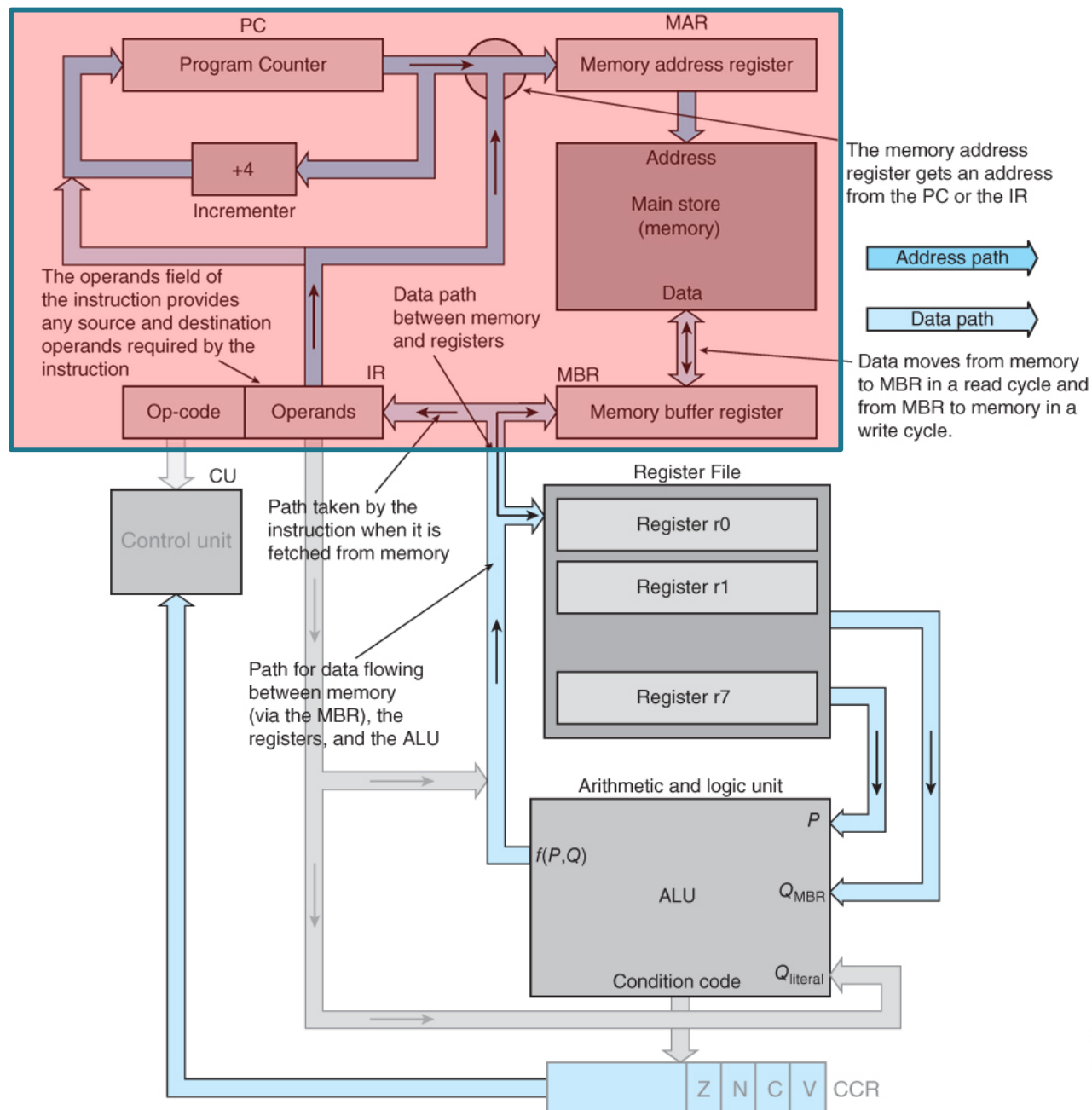
- r0 - r_i** The register file is a *set of general-purpose registers*, e.g., r0, r1, r2, ..., r_i that store temporary (working) data, for example, the intermediate results of calculations, where *i* is typically 8, 16, or 32.
- PC** The *program counter* contains the *address* of the next instruction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

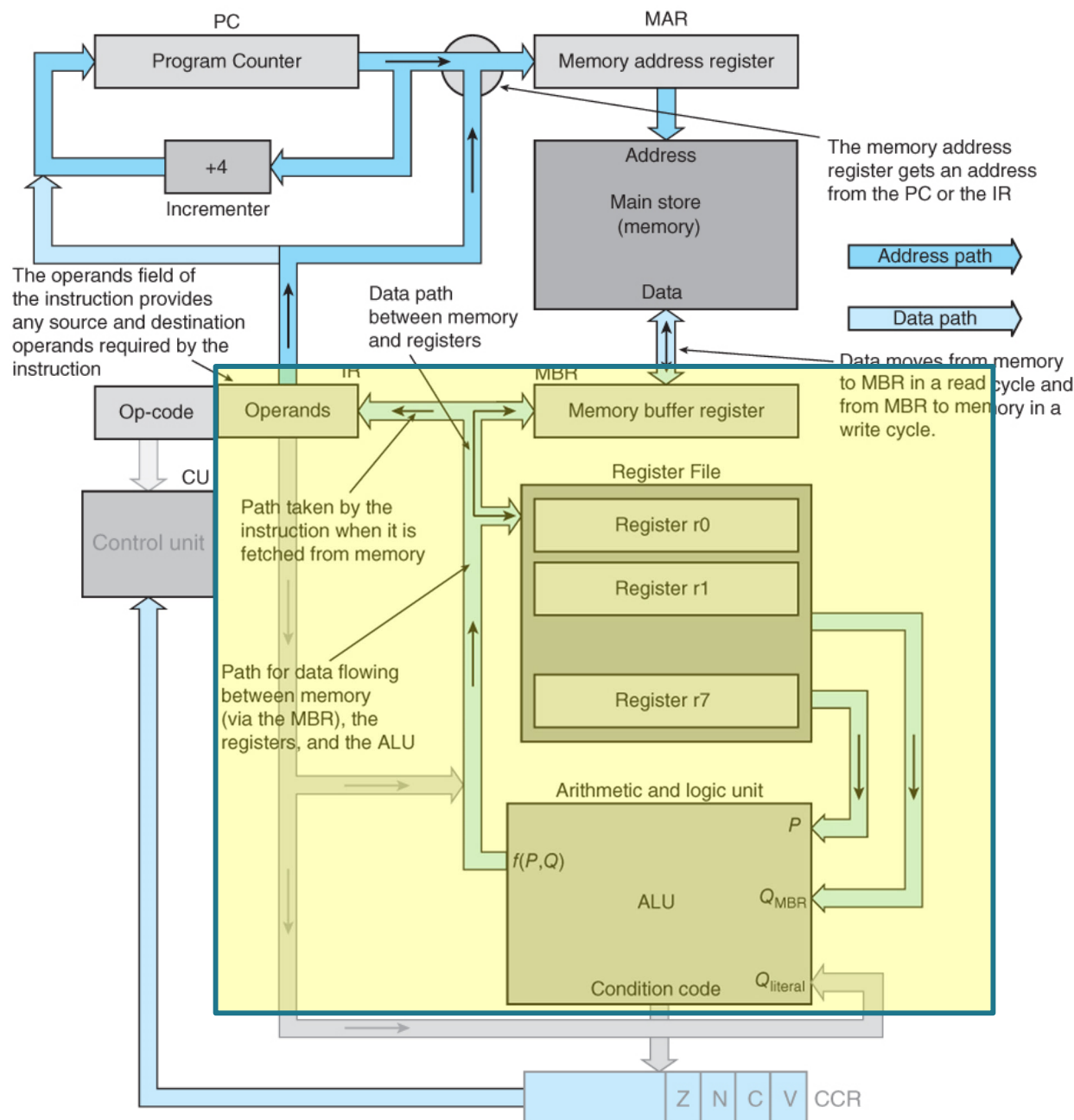
□ We are going to use an ARM processor to introduce assembly language and a modern ISA.

□ *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

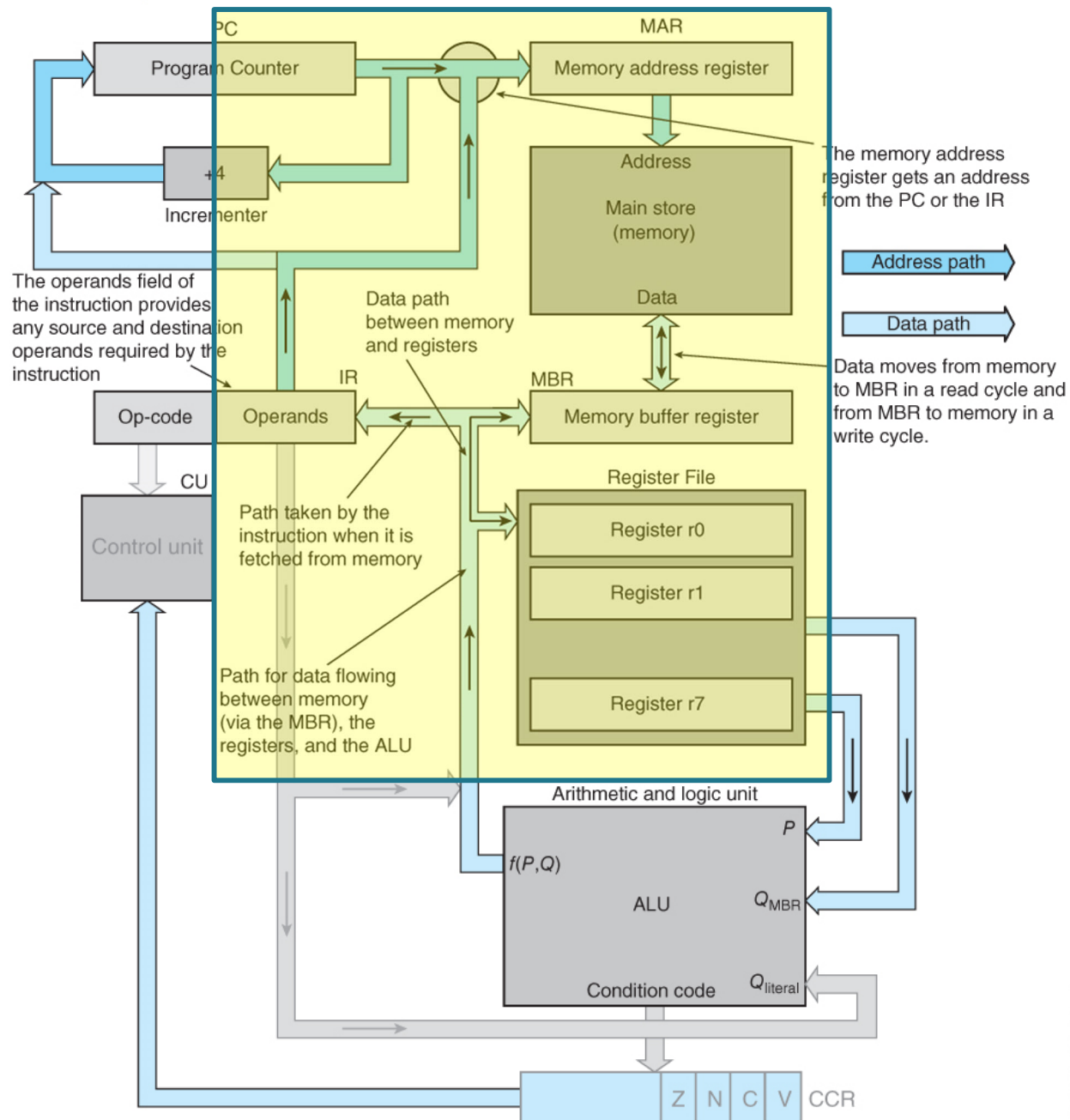
Structure of a Computer

- ❑ 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

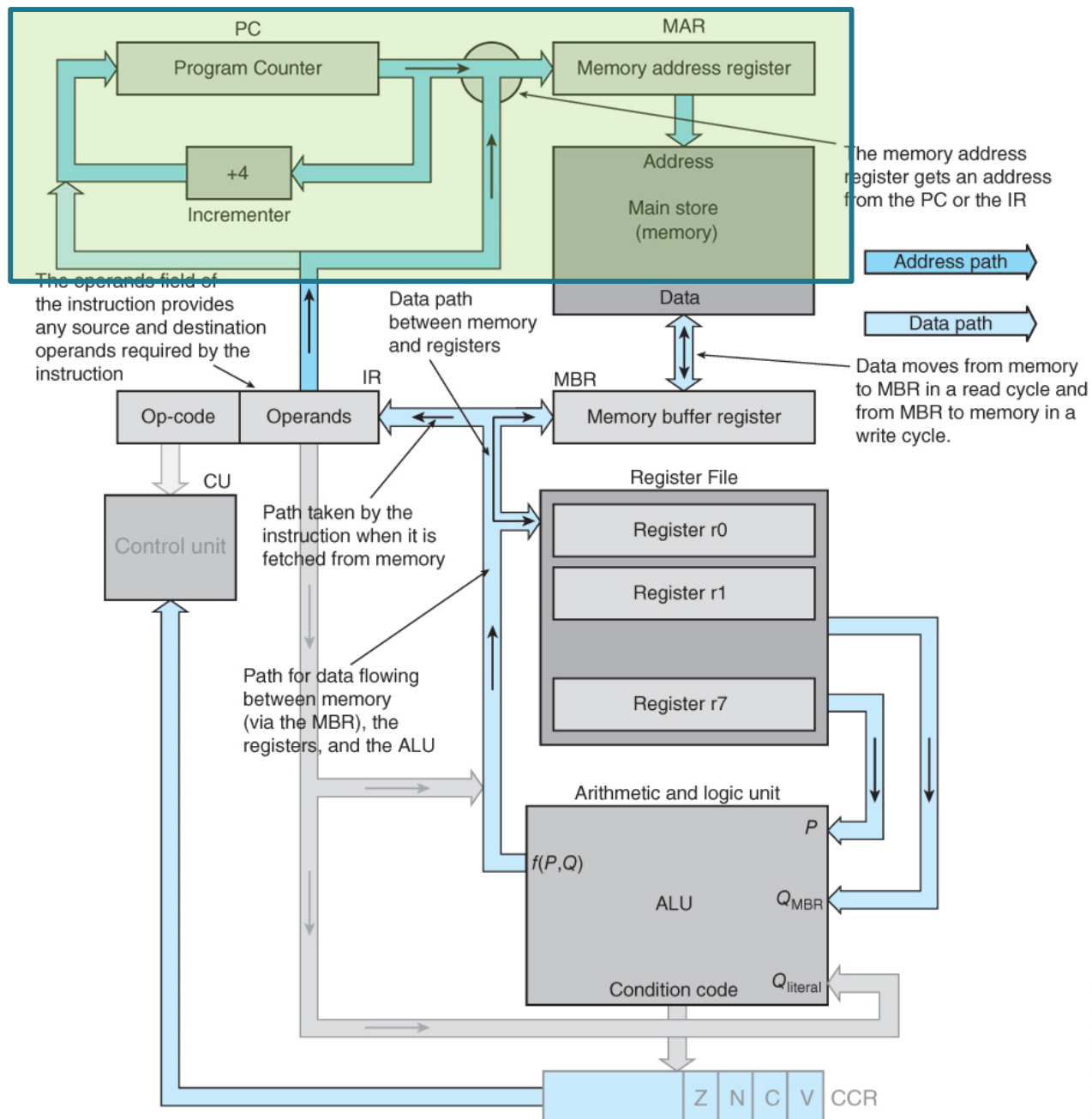
Structure of a Computer

- 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

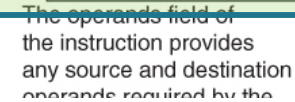
- If the operation requires a memory access (e.g., a load or store), the memory address in the instruction register is sent to the **MAR** and a read or write operation performed.

FIGURE 3.2 Partial structure of a hypothetical stored program machine

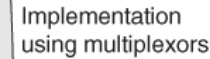
Structure of a Computer

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

FIGURE 3.2



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



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

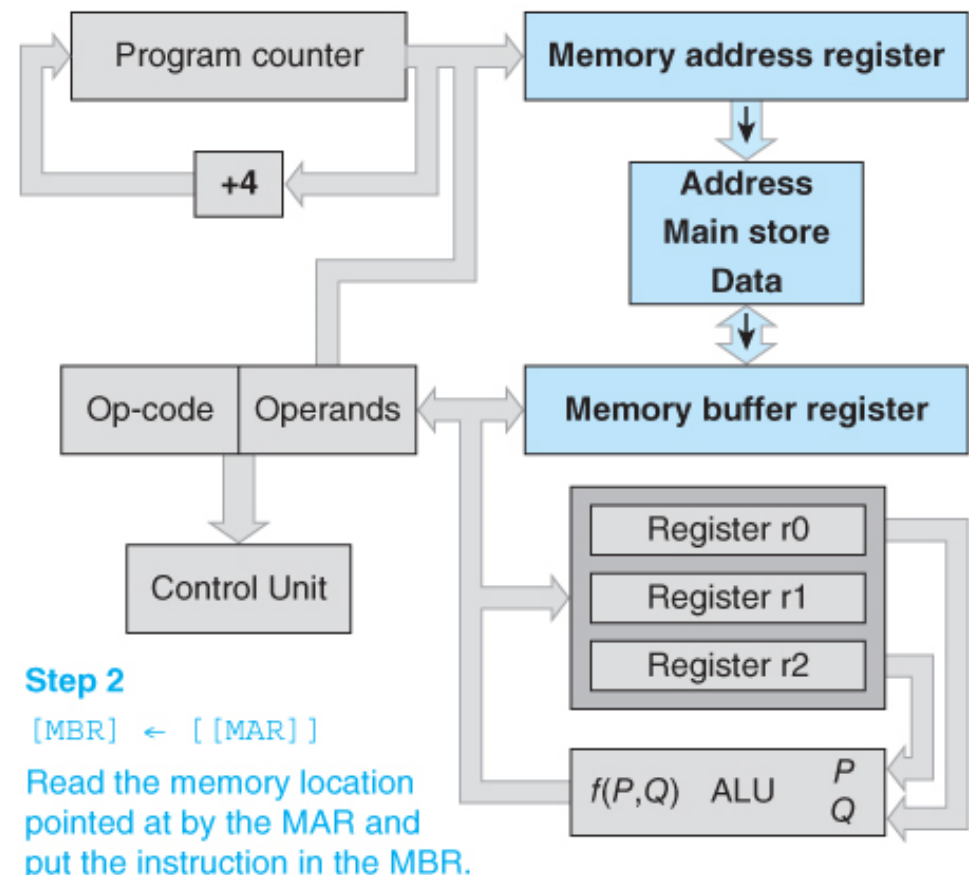
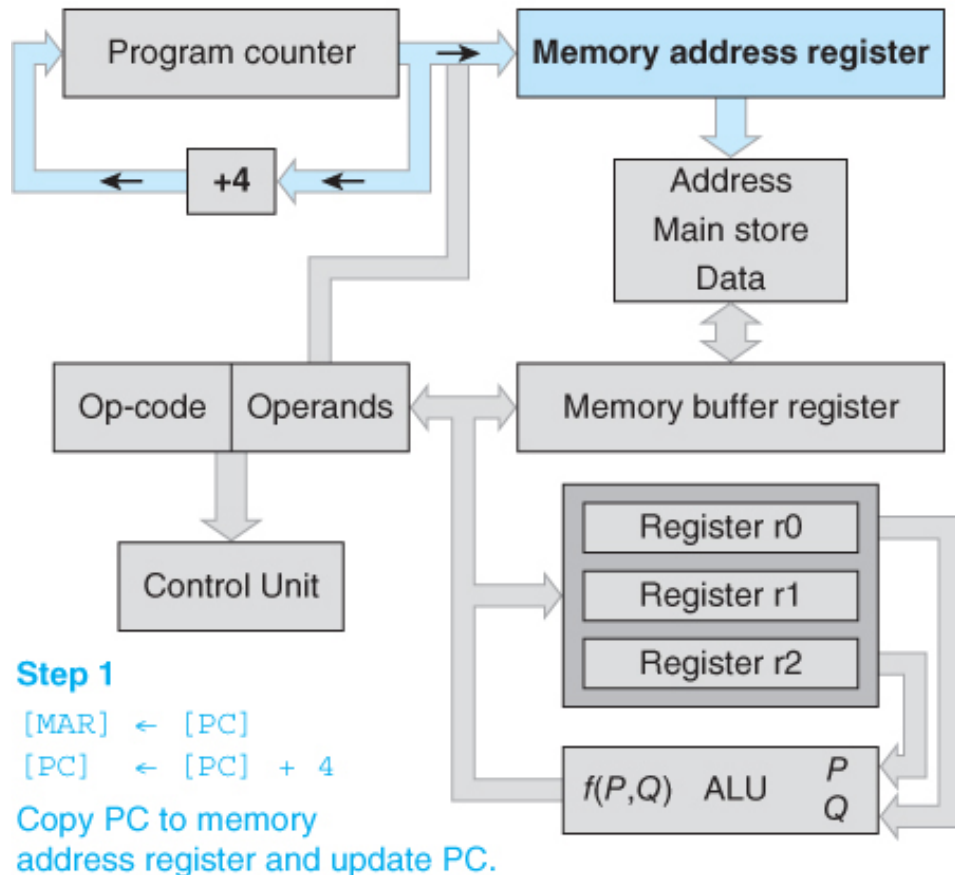
EXECUTE

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

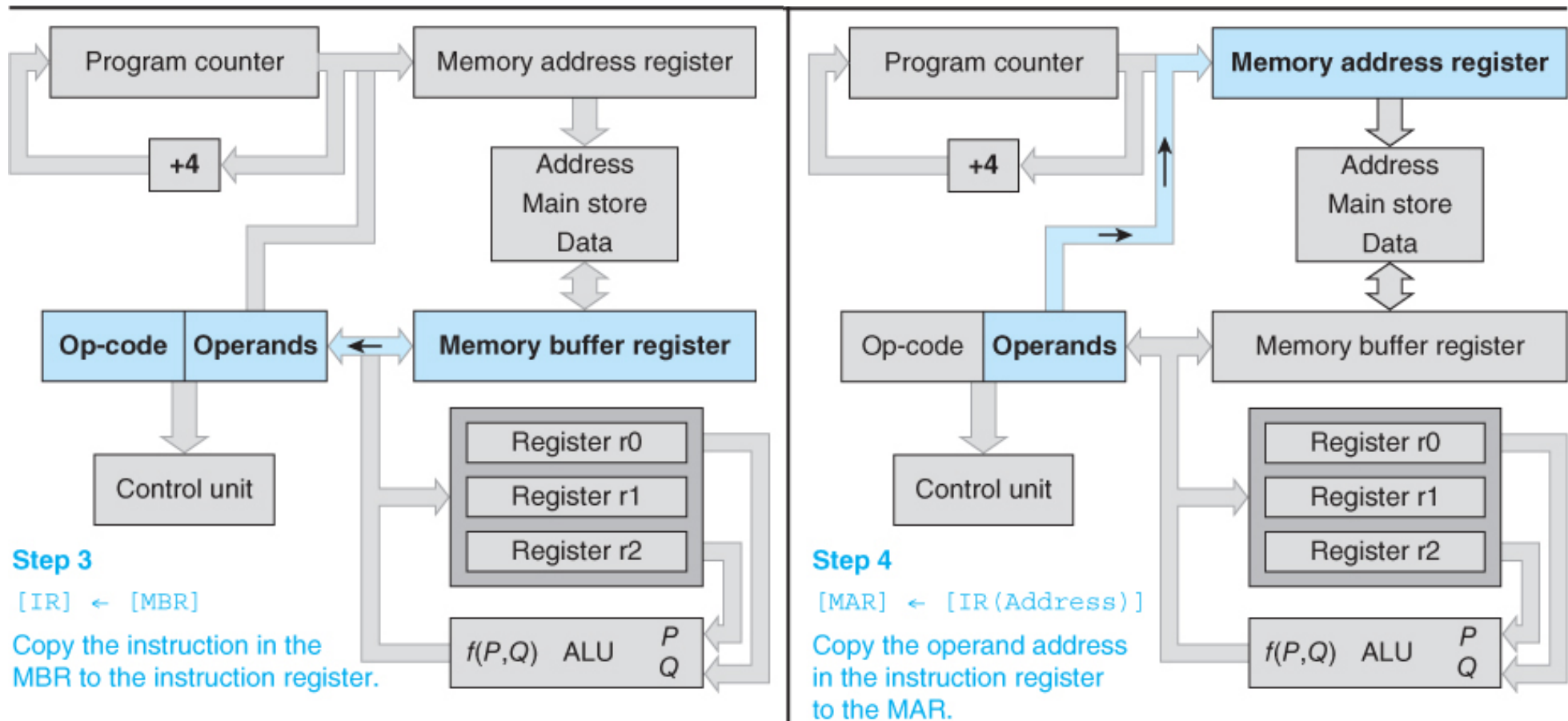
Review Slides 24 and 25 in Chapter 1.

The coming 3 slides show the above steps graphically.

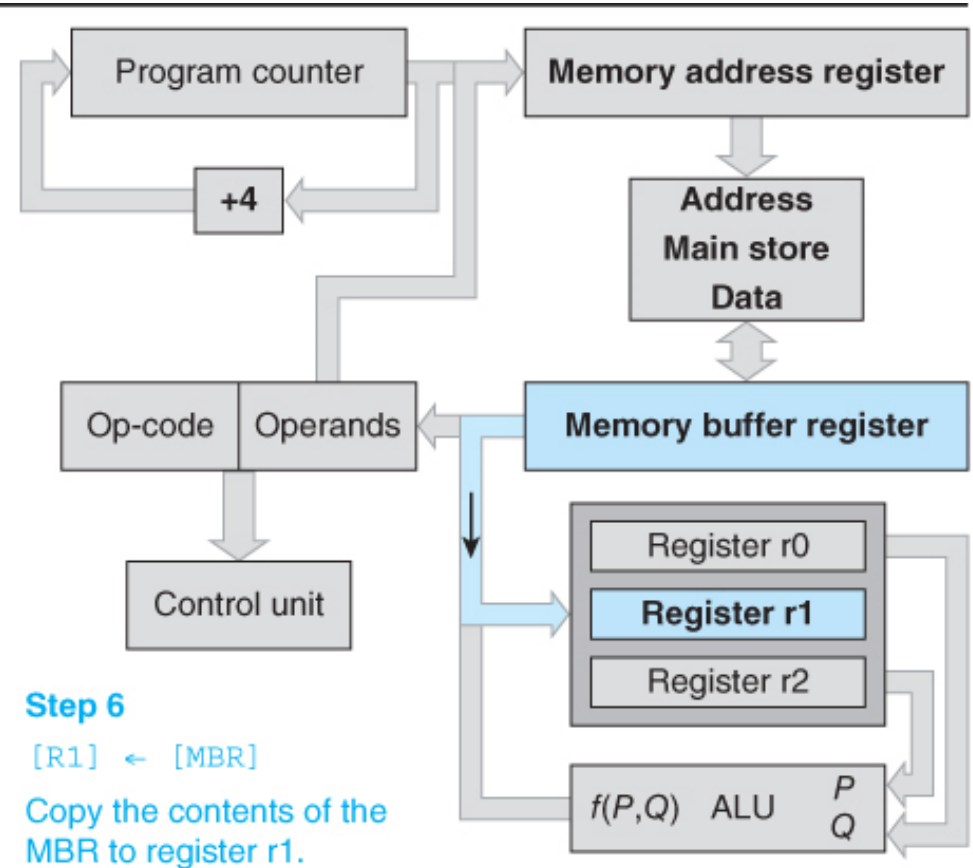
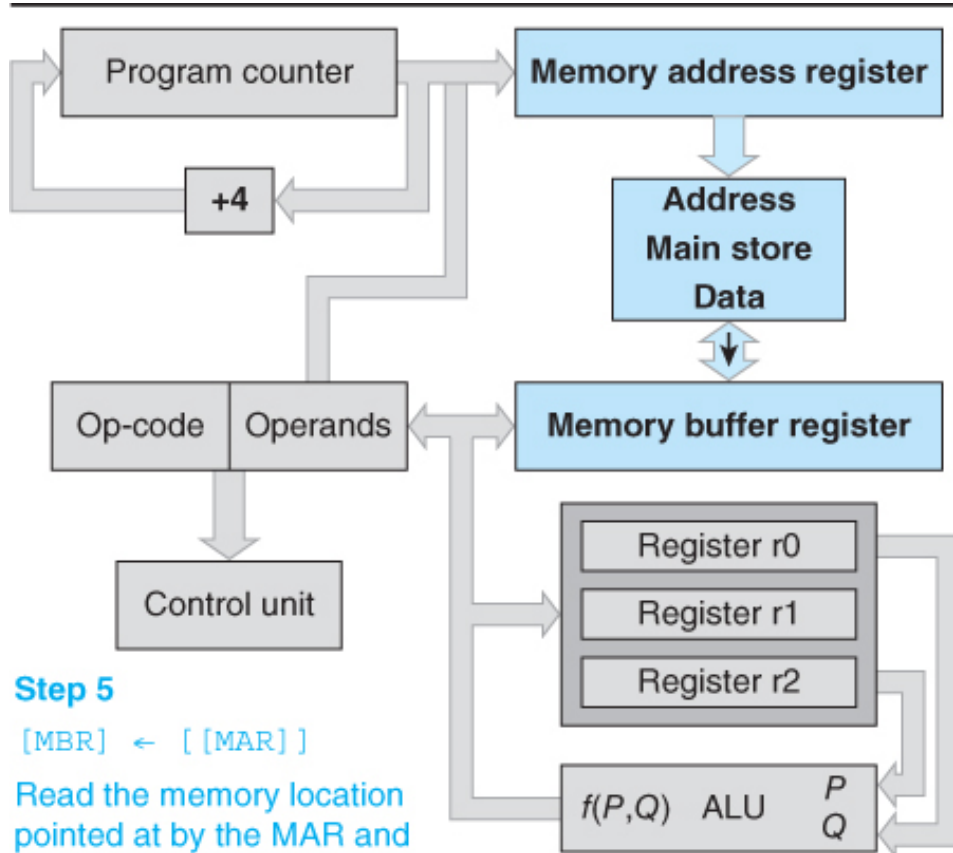
Fetching and Executing an Instruction



Fetching and Executing an Instruction

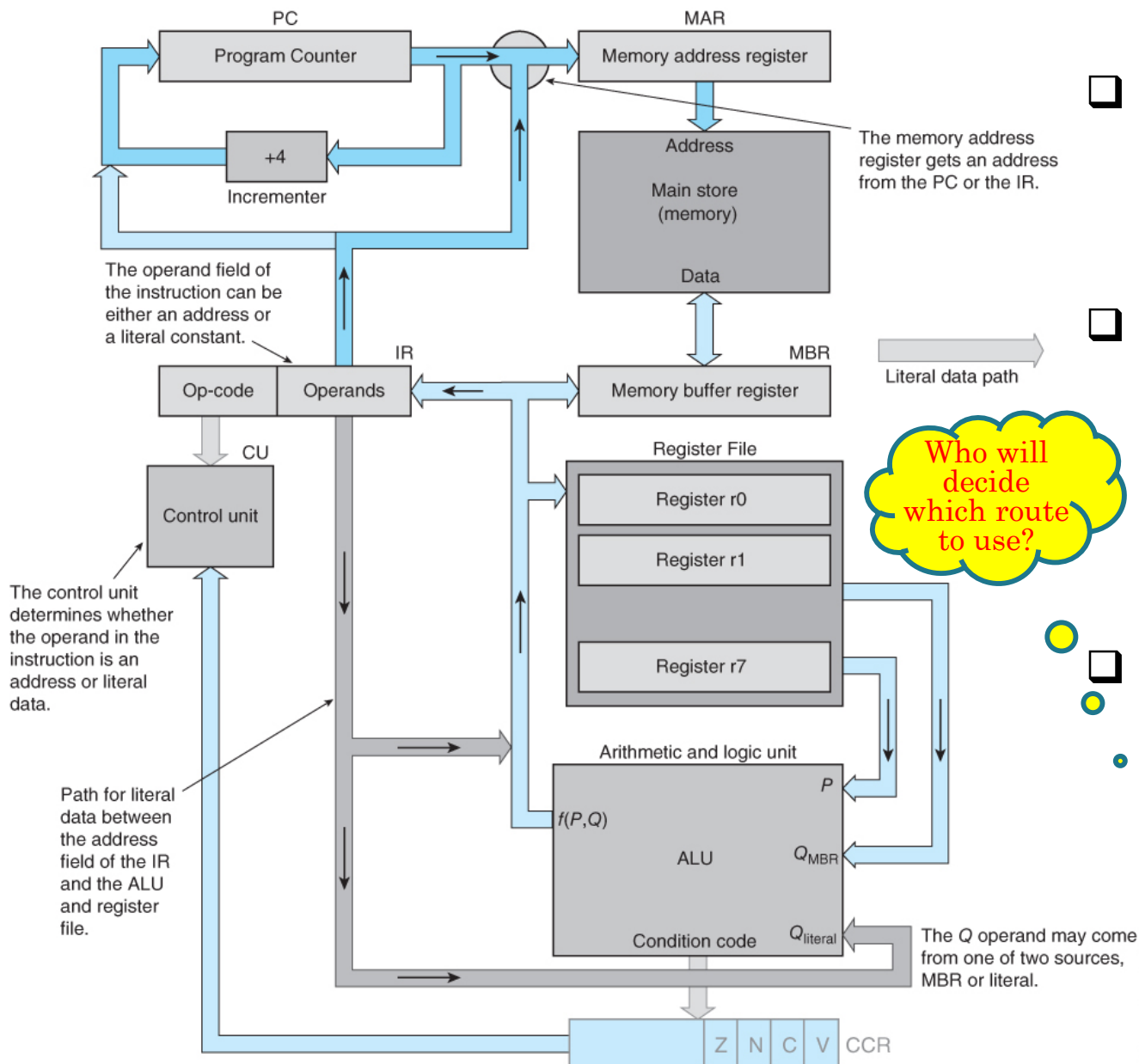


Fetching and Executing an Instruction



Dealing with Constants

FIGURE 3.4 Information paths for literal operands



- ❑ **ADD r0,r1,#25** adds the value 25 (*a.k.a. literal operand*) to the content of r1 and puts the sum in r0
- ❑ 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.
- ❑ In general, the path from the instruction register, **IR**, routes a literal operand to *either* the **register file**, **MBR**, and **ALU**

- ❑ *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.

The diagram illustrates the internal structure and data flow of a computer system. Key components and their interactions are as follows:

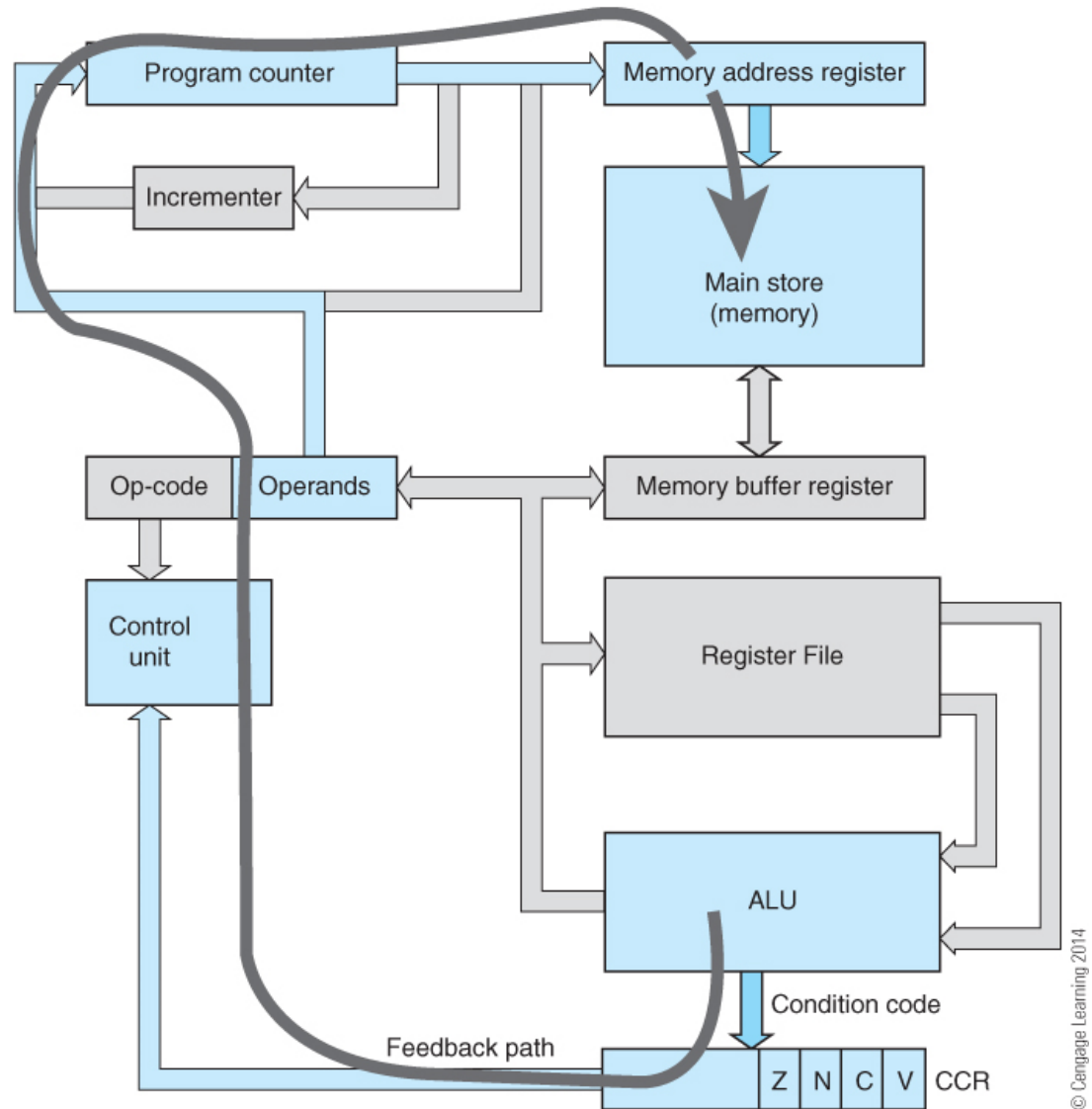
- Program Counter (PC):** Holds the current instruction address. It feeds into the Memory Address Register (MAR) and the Incrementer.
- Memory Address Register (MAR):** Receives the address from the PC and outputs to the Main Store (memory).
- Main Store (memory):** Divided into Address and Data sections. It interacts with the Memory Buffer Register (MBR).
- Memory Buffer Register (MBR):** Acts as a buffer between the Main Store and the Register File and ALU.
- Instruction Register (IR):** Receives instructions from the MBR. It is split into Op-code and Operands.
- Control Unit (CU):** Receives the Op-code from the IR. It controls the Multiplexor and the ALU. A branch control signal from the CU selects the next sequential address from either the incrementer or the address from the IR.
- Incrementer (+4):** Increments the PC value by 4 to determine the next sequential instruction address.
- Multiplexor:** Selects between the incremented address and the address from the IR to update the PC.
- Register File:** Contains registers r0, r1, ..., r7. It provides data to the ALU and receives data from the MBR.
- Arithmetic and Logic Unit (ALU):** Performs operations $f(P, Q)$ on data from the Register File and MBR. It also updates the Condition Code.
- Condition Code:** Provides status bits (Z, N, C, V) to the CCR.
- CCR (Condition Code Register):** Stores the condition code bits (Z, N, C, V) for use by the CU.

The diagram shows the flow of data (blue arrows) and control signals (grey arrows) between these components, illustrating the execution of an instruction.

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.



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
+01000010
-----
```

```
01110101
```

Z = 0, N = 0

C = 0, V = 0

```
11111111
+00000001
-----
```

```
10000000
```

Z = 1, N = 0

C = 1, V = 0

```
01011100
+01000001
-----
```

```
10011101
```

Z = 0, N = 1

C = 0, V = 1

```
11011100
+11000001
-----
```

```
11001101
```

Z = 0, N = 1

C = 1, V = 0

51

+66

117

-1

+1

0

92

+65

-99

-36

-63

-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).