Fetch-Decode-Execute Cycle

Introduction

Modern consumer processors can perform hundreds of billions of instructions per second, and each instruction is interpreted within the fetch-decode-execute cycle [1], [2]. This document describes the process in which the central processing unit receives and executes instructions. The electrical circuitry underlying this process and the implementation of instruction set programming will not be covered here. The document is arranged as follows: First, background information related to the fetch-decode-execute cycle will be defined, followed by the three-stage process itself, and concluding the document is a summary and a brief on pipelining.

Background

Computer instructions exist as binary electrical signals within system memory [3]. Individual instructions come in various formats according to what processor architecture they are intended for but are always composed of an operation code and memory addresses [4], [5]. Operation codes (shortened ***opcode)*** are basic requests such as add, store, or load [6]. The memory addresses function as operands to their opcode [6]. In conjunction, instructions like “add register X to memory location Y” are formed.

Computer programs are algorithms formed out of instruction sequences. After a program has been loaded into system memory, each instruction will pass through the fetch-decode-execute cycle [7]. The program is “running” for the duration of time that its instructions are being executed [7]

Description of the Process

The fetch-decode-execute cycle describes the procedure for obtaining a processor instruction from memory, loading it into the processor, and executing its operation [8]. The following steps have been collected from “Inside the Machine” by John Stokes, “Computer Architecture” by Gerard Blanchet, and “Computer Systems Architecture” by Aharon Yadin.

Stage I: Fetch

This stage involves all the steps for moving instruction data from system memory to the ***central processing unit*** **(CPU)** [8]. *Figure 1* shows the effect this entire stage has on various processor components.

1) Contents of the ***Program Counter*** **(PC)** are stored in the ***Memory Address Register*** **(MAR).** The PC always contains the address of the next instruction to be decoded. The memory address register always contains the memory address for the current read or write operation.

2) The PC is incremented so that it contains the address of the next instruction. The increment used is based upon the system’s instruction size. This step happens early in the cycle to allow for pipelining which is covered in the conclusion of this document.

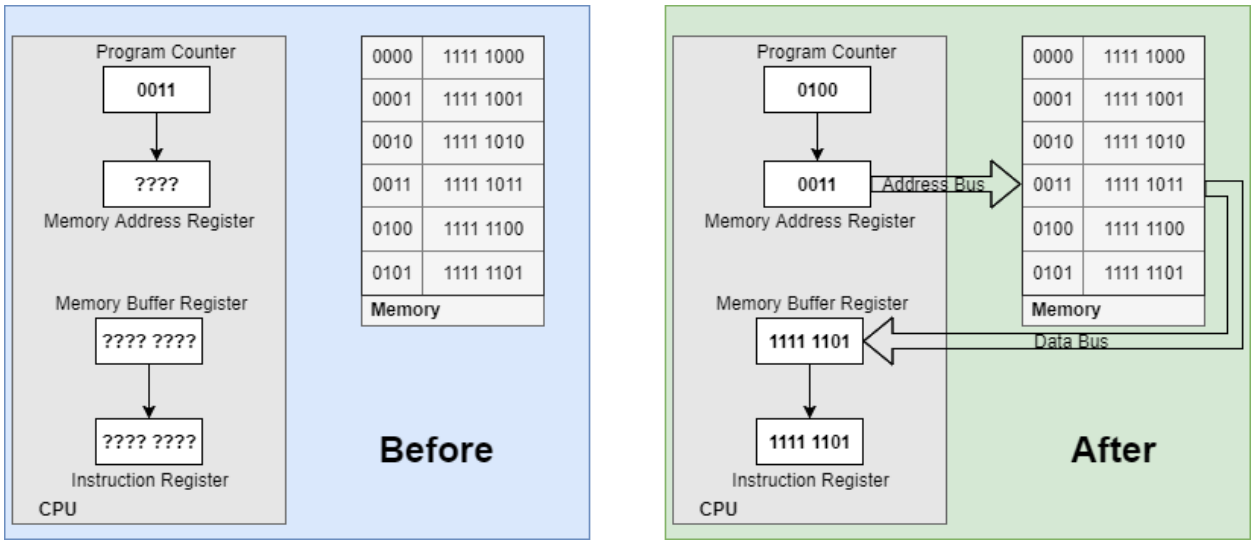
3) The address bus is loaded with the address stored in the MAR and finds it in system memory. Electronic buses connect two physical location by parallel wires, the number of wires denotes

the width of the bus; a bus can transmit as many bits as it is wide [9]. The address bus is a dedicated, unidirectional bus for moving addresses from the CPU to memory.

4) Main Memory data is moved across data bus to the ***Memory Buffer Register (MBR)***. The data is bidirectional and serves to move data between the CPU and memory.

5) Contents of MBR are moved to the ***Instruction Register (IR).*** The instruction is now fully loaded into the CPU.

Figure 1: Effect of Instruction Fetch



This block diagram shows how the state of the CPU changes after the fetch stage. Question marks indicate uninitialized bits, leftover data that was in the register previously. Non-essential processor components are omitted from this diagram for clarity. ©Halston Sellentin 2017

Stage II: Decode

In this stage the instruction type and operands are extracted from the raw instruction data. Operands are moved to their intended processor components which stages them for execution.

1) **Decode Unit (DU)** deconstructs the value stored within the IR, extracting the opcode and operand(s). In *Figure 1* the most significant four bits in the IR represent the opcode '1111' and the least significant four represent the operand '1101'.

2) The decode unit sends the opcode to the ***Control Unit (CU)*** and places the operand in a temporary register on the ***Arithmetic and Logic Unit (ALU).*** The ALU is responsible for performing all calculations in a computer.

3) The control unit matches the opcode to its intended operation and sets the state of the ALU accordingly. The control unit has control lines attached to the ALU and different signals result in different ALU functions.

Stage III: Execute

The CPU is finally ready to perform the given instruction. This stage is localized in the arithmetic and logic unit.

1) The ALU calculates a result. Calculations such as addition and multiplication are done by sending the data through a predefined sequence of logic gates [10]. The opcode determines which sequence to take through the logic gates.

2) The control unit sends the result to an accumulator. For certain computer architectures it is the job of the next instruction to retrieve the result from the accumulator. Other architectures will automatically move the result to another register for storage [1], [3], [8].

Conclusion

The independence of memory and processor allows a cost-effective way for computers to read large programs. The fetch-decode-execute cycle is what facilitates that separation. Without this process, all programs would need to be loaded directly into the size-restricted processor memory. An essential process that builds upon this one is instruction ***pipelining.*** Pipelining allows steps in this process to be performed concurrent to other steps which greatly speeds up the process of reading multiple instructions [11].

References

[1] A. Yadin, “Central Processing Unit” in Computer Systems Architecture, 1st ed. London, England. CRC Press, 2016. [Online]. Available: https://ebookcentral.proquest.com/lib/osu/reader.action?ppg=151&docID=4683317&tm=1510615658723. Ch. 4.

[2] H. Hagedoorn, “Core i7 4700k Processor Review” *Guru 3d*. [Online]. Available: https://www.guru3d.com/articles-pages/core-i7-4770k-review,14.html. [Accessed: November 12, 2017]

[3] G. Blanchet and B. Dupouy, *Computer Architecture: Blanchet/Computer Architecture*. Hoboken, NJ USA. John Wiley & Sons, Inc., 2012. Ch.2, pp. 17–34.

[4] A. González, F. Latorre, and G. Magklis, “Processor Microarchitecture: An Implementation Perspective,” *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 32–33, Dec. 2010.

[5] S. Swanson, “RISC vs CISC,” *Introduction to Computer Hardware*, University of California, San Diego, 2014. [Online]. Available: https://cseweb.ucsd.edu/classes/sp14/cse141-a/Slides/01_ISA-Part%20II-annotated-0429.pdf. [Accessed: November 10, 2017]

[6] K. Irvine, “Floating-Point Processing and Instruction Encoding” in Assembly Language for x86 Processors, 7th ed. New Jersey, USA. Pearson Education Inc., 2015. Ch. 12, pp. 540–547

[7] A. Yadin, “Hardware Architecture” in Computer Systems Architecture, 1st ed. London, England. CRC Press, 2016. [Online]. Available: https://ebookcentral.proquest.com/lib/osu/reader.action?ppg=115&docID=4683317&tm=1510615684631. Ch. 3, pp. 92–93.

[8] J. Stokes “The Mechanics of Program Execution” in Inside the Machine, 1st Edition. California, USA. No Starch Press Inc., 2007. Ch. 2, pp. 26–34

[9] A. Yadin, “Bus” in Computer Systems Architecture, 1st ed. London, England. CRC Press, 2016. [Online]. Available: https://ebookcentral.proquest.com/lib/osu/reader.action?ppg=280&docID=4683317&tm=1510616573958. Ch. 7, pp. 257–260.

[10] D. Page, “Arithmetic and Logic” in A Practical Introduction to Computer Architecture. London, England. Springer Publishing, 2009. Ch. 7, pp. 223–240

[11] G. Blanchet and B. Dupouy, *Computer Architecture: Blanchet/Computer Architecture*. Hoboken, NJ USA. John Wiley & Sons, Inc., 2012. Ch. 10, pp. 207–215.