PROCESSORS

Barry Wilkinson
Department of Computer Science, University of North Carolina at Charlotte, Charlotte, North Carolina, USA

Keywords: CPU, processor, microprocessor, pipeline, microprogramming, RISC, data dependencies, forwarding, superscalar, register renaming, interrupts

Contents

Summary

The focus of this section is on the internal design of the processor (central processing unit, CPU). The processor performs the actions of specified by machine instructions in the program stored in memory. The topics covered include the basic operations of fetching and executing a machine instruction, its implementation, microprogrammed control unit design, and pipelined design. The topic of pipelined design is explored in some detail and includes pipeline hazards, methods to handle procedural dependencies, data dependencies and their control using pipelined interlocks, forwarding, and interrupt handling. Processor design is then continued with superscalar design. Topics here include dual pipelines (briefly), more complex arrangements with multiple functional units as found in more recent processors, out-of-order instruction issue, centralized and distributed instruction windows, register renaming and reorder buffers.

1. Introduction

The processor is the heart of a computer system; it performs the actions specified by the program stored in the memory. This stored program consists of a list of binary encoded machine instructions. Each machine instruction specifies a simple operation which usually operate upon numbers stored in memory or in registers within the processor. The basic ways the machine instructions can be constructed and encoded (instruction formats) is treated elsewhere. Here, we concentrate upon the internal design of the processor to fetch and execute these instructions. There is a very close relationship between the design of the instruction set and the corresponding design of the processor.

Usually, the objective is to design a processor that can execute instructions at the highest possible rate within cost and technological constraints. During the 1970s, there was a trend to incorporate more complex instructions into the instruction set with the belief that this would lead the fastest execution rate. In the late 1970s and early 1980s, this trend was questioned which lead to turnaround towards simpler instruction sets, the so-called reduced instruction set computer (RISC), in the belief that simpler hardware and more powerful compilers would be a better approach to achieve higher performance. However, the legacy of existing processors and their software base continued to be influenced on processor design. Newer models of a processor family need to be compatible with earlier processors of the same family. That is, the newer processors should be able to execute code of older processors, thereby not making the software base obsolete. Ingenious ways were found to keep the complex instruction set while achieving higher performance.

In the following, we will start with a very simple processor design which represents the design of early microprocessors (1970s). We will consider the design of its control unit and the technique of microprogramming. Then, we will progress towards a design used in early pipelined RISC processor design. (Pipelining is a design technique known much earlier and incorporated into supercomputers of the 1960s). We will describe the problems that must be addressed to achieve high performance in pipeline designs and how to improve performance, including executing more than one instruction simultaneously (superscalar designs).

2. Elementary Processor

The internal design of a simple processor is shown in Figure 1, and used here to illustrate the basic operation of a processor. The processor connects to the memory through a set of wires called a bus which carry the data to or from the memory, the address of the memory location being accessed, and control signals that produce the required the actions. The processor contains an ALU (arithmetic and logic unit), a registers file, a control unit and a number of specialized registers. One such register is the instruction register (IR) which holds the instruction that has been fetched for memory to be executed. The ALU performs basic arithmetic and logical operations such as addition and subtraction as specified by the machine instructions. The program counter (PC), also called the instruction pointer (IP), is a register inside the processor holding the address of the next instruction to be executed. The name program counter is unfortunate since it does not count programs; it identifies the next instruction in the program. The control unit orchestrates the actions within the processor by sending signals to the various components.

The operation of this simple processor can be divided into two distinct phases, a fetch cycle and execute cycle. In the fetch cycle, an instruction is obtained from the memory and loaded into the instruction register. Afterwards, the program counter is incremented to point to the next instruction. In the execute cycle, the operation specified by the fetched instruction is performed, including fetching any operands and storing the result. The operations required to fetch and to execute an instruction can be divided into a number of sequential steps. The signals for each step are generated by the control unit of the processor.

The simplest and most common action that the control unit must initiate is a data transfer from one internal register or unit to another internal register or unit. This can be described in a register transfer language (RTL) notation. For example, to transfer the contents of register B to register A in time period T1, we write:

T1: A ← B

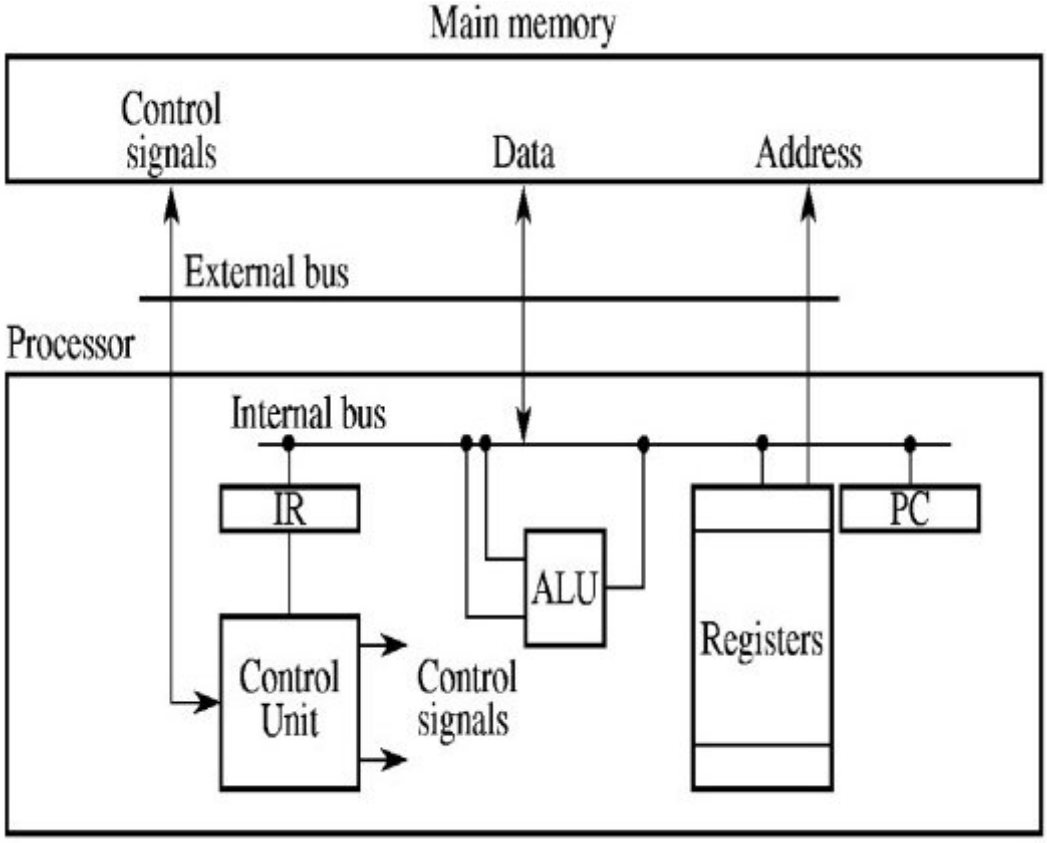

Figure 1. Representative Early Microprocessor Design

This action might be produced by having one signal to select the source (B in this case) and one signal to select the destination (A in this case). A sequence of such steps make up the fetch and execute cycles.

The fetch cycle might be the sequence:

MAR ← PC Select next instruction
Memory read operation
IR ← MDR Load instruction into instruction register
PC ← PC + 4 Increment program counter in preparation for next fetch cycle

where MAR is an internal register holding the address being sent to the memory, and MDR is an internal register holding the data to or from the memory. The PC is

incremented by four in this example because the instructions are assumed to be four bytes each (and the memory is organized so that each byte has a unique address).

The execute cycle will depend upon the fetched instruction, but will also consist of a sequence of elementary steps, each step requiring particular signals to be generated by the control unit. For example, the RISC machine instruction of the form ADD Rd, Rs1, Rs2 (Rd = Rs1 + Rs2) might have the steps:

A ← Rs1 Select first source operand
B ← Rs2 Select second source operand
C ← A + B Perform addition
Rd ← C store result

where A, B, and C are temporary registers within the processor. A and B hold the contents of registers selected as source registers from the register file and C holds the value to be placed in the register selected as the destination register. The selection of source and destination registers will be from the source and destination fields within the machine instruction.

The control unit can be designed using the traditional logic design techniques.This approach leads to a specific logic design interconnected logic gates and flip-flops depending upon the internal arrangements of the processor and the instruction set (a so-called a random logic approach). To implement very complex instructions especially, an alternative more structured approach called microprogramming may be better. Here, each step is encoded into a binary pattern called a microinstruction. A sequence of these microinstructions is formed for each machine instruction. The sequence is stored in a high speed memory called a control memory which is within the control unit of the processor. The sequence of microinstructions is known as microprogram (or microcode) and one sequence must be performed to execute each machine instruction.

The general arrangement of a microprogrammed control unit is shown in Figure 2. The next machine instruction to be executed is read from the memory into the instruction register by a instruction fetch microprogram (or by dedicated logic). The fetched machine instruction identifies the location in the control memory of the first microinstruction of the execute sequence for that machine instruction, and this sequence is then fetched from the control memory and performed.

Microprograms are very much like machine instruction programs and some of the same techniques can be applied to both. For example, a microinstruction program counter can be used to hold the address of the next microinstruction to be executed, just as a machine instruction program counter holds the address of the next machine instruction to be executed. The execution sequence can be altered by conditions occurring within or outside the processor. In particular, conditional branch machine instructions may require microinstruction sequences that are altered by conditions indicated in the processor condition code register. This suggests the use of conditional branch microinstructions. In these microinstructions, the branch address (the address of the next microinstruction, should the conditions be met) is provided in a next address field of the microinstruction. Microinstruction procedures (subroutines) can also be created to reduce the size of the

microprogram. A microinstruction stack is then used within the control unit to hold the return addresses of machine instruction procedures.

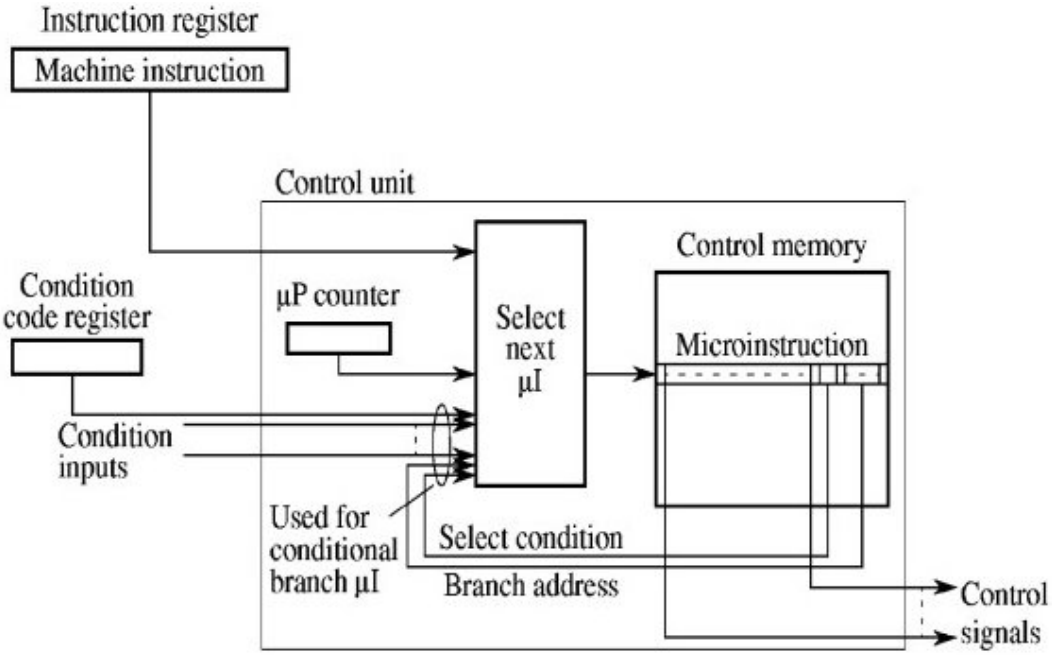

Figure 2.Microprogrammed Control Unit

In general, microinstructions are divided into two fields, one known as a micro-order identifying the signals and actions to be generated, and one, the next address field, used only for conditional microinstructions, giving the address of the next microinstruction to be executed if some condition exists. The condition is specified in another field of the microinstruction. (It is also possible to provide a next address field for all microinstructions instead of using a microinstruction program counter; this approach was used in the original microinstruction design.)

The micro-order field can have one bit for each signal to be generated, binary encoded fields, or a combination. A two-level approach is also possible, in which a short microinstruction points to a set of much longer nanoinstructions held in another control memory.

Providing one bit for each signal is called the horizontal micro-order format. If there were 100 possible signals, there would be 100 bits. To generate a particular signal, the corresponding bit would be set to a 1. More than one signal could be generated simultaneously by setting more bits to a 1. For example, to perform a data transfer from one register to another register, one bit could select the source register and another bit could select the destination register.

Encoding the bits into fields is called the vertical micro-order format and is used to reduce the number of bits. Bits are formed into groups specifying signals that cannot be activated together. For example, for a basic register transfer operation, only one

destination register can be selected at a time. If there were 15 possible destination registers, 4 bits could be used to select the destination (one pattern, say 0000, might be reserved for no destination). Similarly, another four-bit field could select the source register if there were 15 possible source registers.

The horizontal format is the most flexible but results in a long microinstruction whereas the vertical format is more efficient but requires logic to decode the patterns and hence the speed of operation is reduced. Also, mutually exclusive signals must be identified prior to designing the microinstruction. Usually, a combination of both formats is used.

The concept of a microprogrammed control unit was first suggested by Wilkes in the early 1950s but was not put into commercial practice until the 1960s. The approach is particularly convenient for implementing complex instructions and was popular in the 1970s but lost favor since the early 1980s with the advent of processors having simpler instruction sets. However, microprogramming can still be found to deal with complex instructions.

3. Pipelined Processor Design

3.1. Technique

Pipelining is a technique used to increase the speed of processors. It was first employed in high performance processors in the 1960s and now can found in most, if not all, processors. The techniques rely on being able to divide the actions of the processor into a number of sequential steps, each of which is implemented with a separate unit (pipeline stage). The mechanism can be compared to a conveyor belt assembly line in a factory in which products are in various stages of completion. Each product is assembled in stages as it passes along the assembly line. And just as an assembly line leads to increased production, the pipeline technique can lead to substantial increase in the speed of execution of a series of instructions. It does rely on a series of instructions and ideally each stage or unit should require the same time or else the speed will be limited by the slowest unit.

The simplest processor pipeline would be to have two units, one to fetch an instruction from memory and another to execute the fetched instruction. With this two-stage pipeline, the fetch unit would be fetching the *n*th instruction at the same time as the execute unit is executing the (*n* - 1)th instruction.

Figure 3(a) shows a 4-stage pipeline. This pipeline consists of one unit to fetch instructions, one to fetch the operands, one to perform the operation specified by the instruction, one to store the result. Figure 3(b) shows the timing of this pipeline using a space-time diagram which depict what each unit is doing in each pipeline cycle. (Another common form of space-time diagram depicts the units being used for each instruction horizontally, but for simplicity we will keep to the form shown here.) In most pipelines, the information that passes from one unit to the next is held latches between the units and all the latches are synchronized to a common clock; such detail is not shown in the figures here.)

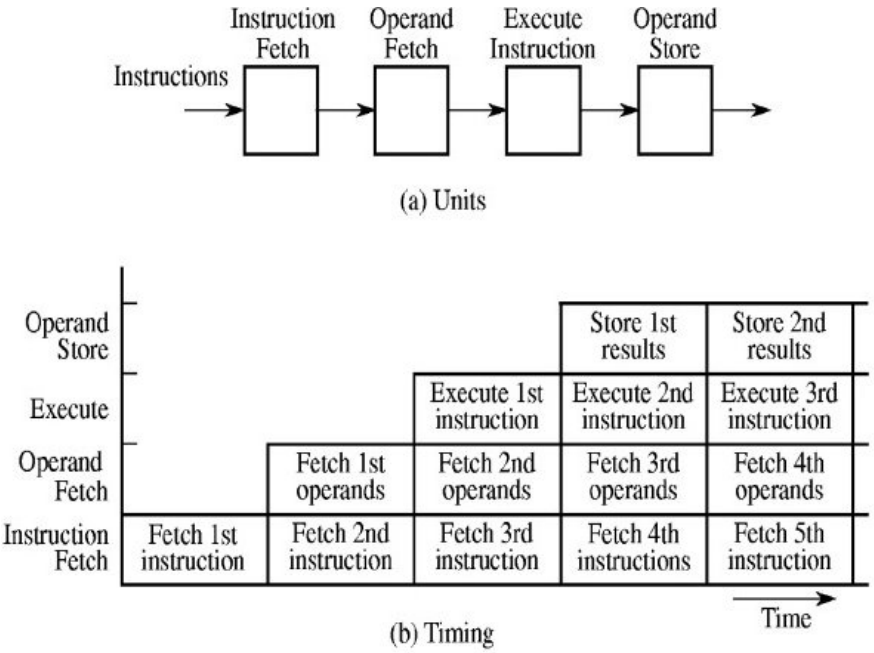

Figure 3. Four-Stage Instruction Pipeline

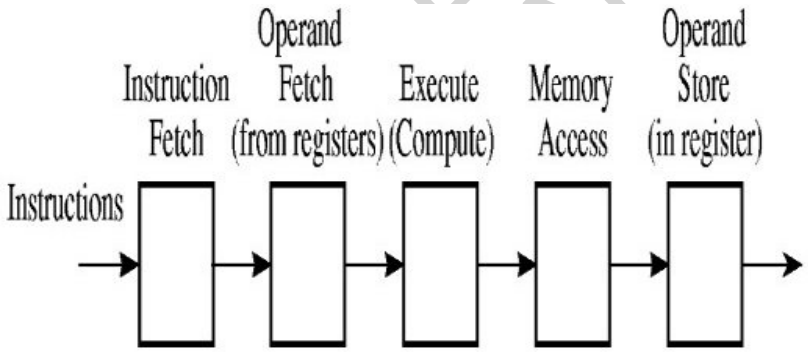

Figure 4. Five-Stage Instruction Pipeline with One Stage for Accessing Memory

The 4-stage pipeline described here would be suitable when both the source operands and the results are held in registers. To process instructions that access memory, units are necessary to compute the effective memory address and access the memory. There are many possible pipeline structures. In a RISC processor with only load and store instructions that access memory, a single additional memory access unit can be placed after the execute unit as shown in Figure 4. The execute unit is now used to compute the effective address for memory reference instructions, in addition to performing arithmetic/logic operations for register-register instructions. The operand store unit is not used for memory write instructions. This pipeline structure appeared in early RISC processors. More recent RISC processors usually have more pipeline stages. More than one stage can be provided for operations that require more time rather than extend the clock period. For example, memory access might require more time. Suppose it requires two clock periods. Two stages could be allocated to the memory accesses. (Most

processors use cache memory as described in Chapter *Computer Architectures* and *Memory Systems: Overview and Trends* and this is assumed for these memory accesses.)

3.2. Instruction Pipeline Hazards

Pipeline structures work well if is a sequence of tasks to be performed in order, and so far we have assumed that there is no interaction between pipeline units other than passing the results of one unit on to the next unit. However, there is are situations may require the pipeline to hesitate (stall) which of course we would like to avoid or reduce as it impacts directly on the speed of the system. In this section, the various reasons for pipeline stalls and methods to overcome them where possible will be outlined.

There are three major causes for a hesitation of an instruction pipeline (pipeline hazards), namely resource conflicts, procedural dependencies (caused notably by branch instructions), and data dependencies.

-
-
-

Bibliography

Hill M. D. Jouppi N. P. and Sohi G. S., editors, (2000). *Readings in Computer Architecture*, 717 pp. San Francisco, CA: Morgan Kaufmann Publishers. [This is a recent collection of important historically significant papers.]

Patterson D. A. and Hennessy J. L. (1996). *Computer Architecture A Quantitative Approach 2nd edition*, 760 pp. San Francisco, CA: Morgan Kaufmann Publishers. [This is the classic and definitive text on RISC design.]

Shriver B. and Smith B. (1998). *The Anatomy of a High-Performance Microprocessor A Systems Perspective*, 545 pp. Los Alamitos, CA: IEEE Computer Society Press. [This unique book details the design of a high-performance processor and has a large amount of supporting materials on a CD-ROM.]

Sima D. Fountain T. and Kacsuk P. (1997). *Advanced Computer Architecture A Design Space Approach*, 766 pp. Harlow, England: Addison-Wesley. [This text has an excellent section on high performance processor design (Part II).]

Wilkinson B. (1996). *Computer Architecture Design and Performance 2nd edition*, 463 pp. London: Prentice Hall. [This text is written by the author and contains additional information on processor design.]