Operating Systems

Todd Davies

September 28, 2014

Introduction

Operating systems provide an interface for computer users that permits them to gain access without needing to understand how the computer works. The software needed to achieve this is complex and this course introduces students to some of the details of design and implementation.

Aims

This course unit introduces students to the principles of operating system design and to the prevailing techniques for their implementation. The course unit assumes that students are already familiar with the structure of a user-program after it has been converted into an executable form, and that they have a rudimentary understanding of the performance trade-offs inherent in the choice of algorithms

and data structures. Pertinent features of the hardware-software interface are described, and emphasis is placed on the concurrent nature of operating system activities. Two concrete examples of operating systems are used to illustrate how principles and techniques are deployed in practice.

Additional reading

Operating system concepts (8th edition)	Silberschatz, Abraham and Peter Baer Galvin and Greg Gagne	2009
Operating system concepts with Java (8th edition)	Silberschatz, Abraham and Peter Baer Galvin and Greg Gagne	2010

Contents

1 Recap of COMP121 & COMP151

There material in both the *Fundamentals of Computer Architecture* and the *Fundamentals of Computer Engineering* courses in the first year provides a good base for the course this year. The following re-visits that material and builds upon it.

1.1 Datapath and Control

From the point of view of the CPU all data is of a fixed size, the length of one word. Each word is usually moved around the architecture of the computer in a bit-parallel manner, that is to say that there are at least the same number of wires in a bus between any two components in the system as there are bits in the word.

The individual operations on each bit inside a word when the CPU performs an operation on the whole word are usually identical. This results in a very regular datapath with lots of duplicated (usually by as many times as the word length) hardware logic.

Control logic is derived after the datapath has been conceived. It governs which operation is performed at what time, and is different for each instruction in the instruction set.

A typical example of control logic might be to control the enable pin on a binary adder. The datapath will direct bits to the adder all the time, but the control logic will determine if the result is sent forward.

1.2 The MU0 Instruction Set Architecture

The MU0 is a very simple 16 bit word architecture, and as a result, the instruction set is also very simple. Each instruction can address one memory location, and consists of four bits for the instruction (allowing sixteen instructions to be coded) and twelve bits for the memory address as illustrated in Figure 1. Since we can only store twelve bits of memory address in the instruction, and the architecture is very simple, the system has 2^{12} words of memory, which is equivalent to 8 kB.

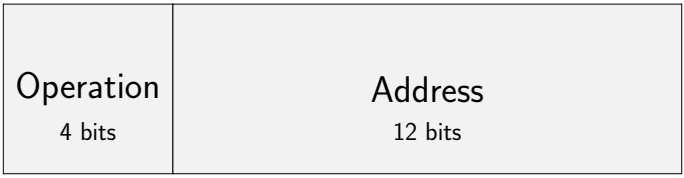

Figure 1: A generic MU0 instruction

The MU0 has two programmer visible registers, the Program Counter and the Accumulator. The Program Counter stores the address in memory of the next instruction to be executed, thus being twelve bits long. The Accumulator is sixteen bits long, and stores the result of the last arithmetic operation.

The instructions that the MU0 understands are listed in Table 1.

Op Code	Mnemonic	Description
0	LDA $[op]$	$[op] \rightarrow Acc$
1	STO $[op]$	$Acc \rightarrow [op]$
2	ADD $[op]$	$Acc = Acc + [op]$
3	SUB $[op]$	$Acc = Acc - [op]$
4	JMP $[op]$	$PC = S$
5	JGE $[op]$	If $Acc >= 0$ then $PC = S$
6	JNE $[op]$	If $Acc \neq 0$ then $PC = S$
7	STP	Stop

Table 1: The MU0 instruction set

1.3 Maintaining Processor State

If the execution cycle of the MU0 was somehow disrupted, say because of an interrupt call, it would be handy to save the state of the processor before switching to a different task (e.g. running the interrupt handler).

The way to do this is to save the registers in memory, doing the other task, and then reloading them when it's time to resume execution of the program.

1.4 The Fetch Execute Cycle

The fetch-execute cycle describes how a CPU executes instructions. First, the next instruction is fetched from memory (at the address pointed to by the PC), then the instruction is executed. Since some instructions access memory (such as load and store), and we can only do one memory access per clock cycle, one fetch-execute cycle takes two clock cycles,

one for fetching, and one for execution.

1.4.1 Fetching instructions

Fetching is an operation that is the same for all instructions. First memory addressed by the PC is read and stored into the Instruction Register (IR). This is a 16 bit internal register that isn't visible to programmers. Once this has occurred, the PC is incremented. This means that the RAM must be able to send a word directly to the instruction register, so a datapath must be in place to allow this.

1.4.2 Executing instructions

It is obvious that different instructions will have different paths of execution within the processor, and will have different effects on components within the system.

`JMP` In order to execute the `JMP` instruction, the last twelve bits are read from the instruction register and transferred over to the PC. This means that there must be a datapath from the bottom twelve bits of the IR to the PC.

`STA` When `STA` is executed, the bottom twelve bits in the IR are used direct the contents of the accumulator to a location in memory. To do this, we need a datapath from the bottom twelve bits of the IR to the part of the RAM that takes addresses, and from the PC to the part of the RAM that takes data.

`ADD` To perform the `ADD` instruction, we need to fetch the bottom twelve bits of the IR and send it to the RAM. The result should be fed into the adder along with the contents of the accumulator. The result of the calculation should be sent to the accumulator. To do this, we need datapaths from the accumulator to the ALU, the RAM to the ALU and finally from the ALU to the accumulator.

Control Signals whenever two separate components within the system interact. For example, every time the CPU loads a word from the RAM, a control signal must be sent to say 'load', and every time the `ADD` command it executed, the ALU must be sent a control signal to say 'add' as opposed to subtract or shift.

Timing Timing is very important when executing the instructions. If the result of a load from RAM hasn't yet returned, but the control signal to the ALU to add is sent, then the wrong result will almost certainly occur! In order for everything to run smoothly, the critical path for each operation must be worked out, and time allowed for signals to propagate through even the longest critical path.

1.4.3 Deriving the datapaths from the operation of instruction

In order to produce a working processor, we need to look at all the instructions that can be executed by the processor, and examine what datapaths and control signals they require

to work. Only when we have this information can we begin to actually design the hardware on the CPU.

Note:

Note that data going to one destination can only go to one source, so if you want multiple components to be able to send data to one other component, then you must use a multiplexer with control signals in order to achieve this.