# APPENDIX C
# TOPICS IN COMPUTER ORGANIZATION

**William Stallings**




Supplement to
Operating Systems, Eighth Edition
Pearson 2014
http://williamstallings.com/OperatingSystems/

This appendix provides additional details to supplement Chapter 1.

## C.1 PROCESSOR REGISTERS

A processor includes a set of registers that provide memory that is faster and smaller than main memory. Processor registers serve two functions:

- **User-visible registers:** Enable the machine or assembly language programmer to minimize main memory references by optimizing register use. For high-level languages, an optimizing compiler will attempt to make intelligent choices of which variables to assign to registers and which to main memory locations. Some high-level languages, such as C, allow the programmer to suggest to the compiler which variables should be held in registers.
- **Control and status registers:** Used by the processor to control the operation of the processor and by privileged OS routines to control the execution of programs.

There is not a clean separation of registers into these two categories. For example, on some processors, the program counter is user visible, but on many it is not. For purposes of the following discussion, however, it is convenient to use these categories.

### User-Visible Registers

A user-visible register may be referenced by means of the machine language that the processor executes and is generally available to all programs, including application programs as well as system programs. Types of registers that are typically available are data, address, and condition code registers.

**Data registers** can be assigned to a variety of functions by the programmer. In some cases, they are general purpose in nature and can be used with any machine instruction that performs operations on data. Often, however, there are restrictions. For example, there may be dedicated registers for floating-point operations and others for integer operations.

**Address registers** contain main memory addresses of data and instructions, or they contain a portion of the address that is used in the calculation of the complete or effective address. These registers may themselves be general purpose, or may be devoted to a particular way, or mode, of addressing memory. Examples include the following:

- **Index register:** Indexed addressing is a common mode of addressing that involves adding an index to a base value to get the effective address.
- **Segment pointer:** With segmented addressing, memory is divided into segments, which are variable-length blocks of words.[1] A memory reference consists of a reference to a particular segment and an offset within the segment; this mode of addressing is important in our

[1] There is no universal definition of the term *word*. In general, a **word** is an ordered set of bytes or bits that is the normal unit in which information may be stored, transmitted, or operated on within a given computer. Typically, if a processor has a fixed-length instruction set, then the instruction length equals the word length.

discussion of memory management in Chapter 7. In this mode of addressing, a register is used to hold the base address (starting location) of the segment. There may be multiple registers; for example, one for the OS (i.e., when OS code is executing on the processor) and one for the currently executing application.

- **Stack pointer:** If there is user-visible stack[2] addressing, then there is a dedicated register that points to the top of the stack. This allows the use of instructions that contain no address field, such as push and pop.

For some processors, a procedure call will result in automatic saving of all user-visible registers, to be restored on return. Saving and restoring is performed by the processor as part of the execution of the call and return instructions. This allows each procedure to use these registers independently. On other processors, the programmer must save the contents of the relevant user-visible registers prior to a procedure call, by including instructions for this purpose in the program. Thus, the saving and restoring functions may be performed in either hardware or software, depending on the processor.

## Control and Status Registers

A variety of processor registers are employed to control the operation of the processor. On most processors, most of these are not visible to the user.

---

[2] A stack is located in main memory and is a sequential set of locations that are referenced similarly to a physical stack of papers, by putting on and taking away from the top. See Appendix P for a discussion of stack processing.

Some of them may be accessible by machine instructions executed in what is referred to as a control or kernel mode.

Of course, different processors will have different register organizations and use different terminology. We provide here a reasonably complete list of register types, with a brief description. In addition to the MAR, MBR, I/OAR, and I/OBR registers mentioned in Chapter 1 (Figure 1.1), the following are essential to instruction execution:

- **Program counter (PC):** Contains the address of the next instruction to be fetched
- **Instruction register (IR):** Contains the instruction most recently fetched

All processor designs also include a register or set of registers, often known as the program status word (PSW), that contains status information. The PSW typically contains condition codes plus other status information, such as an interrupt enable/disable bit and a kernel/user mode bit.

**Condition codes** (also referred to as *flags*) are bits typically set by the processor hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set following the execution of the arithmetic instruction. The condition code may subsequently be tested as part of a conditional branch operation. Condition code bits are collected into one or more registers. Usually, they form part of a control register. Generally, machine instructions allow these bits to be read by implicit reference, but they cannot be altered by explicit reference because they are intended for feedback regarding the results of instruction execution.

In processors with multiple types of interrupts, a set of interrupt registers may be provided, with one pointer to each interrupt-handling routine. If a stack is used to implement certain functions (e.g., procedure call), then a stack pointer is needed (see Appendix 1B). Memory management hardware, discussed in Chapter 7, requires dedicated registers. Finally, registers may be used in the control of I/O operations.

A number of factors go into the design of the control and status register organization. One key issue is OS support. Certain types of control information are of specific utility to the OS. If the processor designer has a functional understanding of the OS to be used, then the register organization can be designed to provide hardware support for particular features such as memory protection and switching between user programs.

Another key design decision is the allocation of control information between registers and memory. It is common to dedicate the first (lowest) few hundred or thousand words of memory for control purposes. The designer must decide how much control information should be in more expensive, faster registers and how much in less expensive, slower main memory.

## C.2 INSTRUCTION EXECUTION FOR I/O FUNCTIONS

This section supplements the information in Section 1.3.

Data can be exchanged directly between an I/O module (e.g., a disk controller) and the processor. Just as the processor can initiate a read or write with memory, specifying the address of a memory location, the processor can also read data from or write data to an I/O module. In this latter case, the processor identifies a specific device that is controlled by a particular I/O module. Thus, an instruction sequence similar in form to that

of Figure 1.4 could occur, with I/O instructions rather than memory-referencing instructions.

In some cases, it is desirable to allow I/O exchanges to occur directly with main memory to relieve the processor of the I/O task. In such a case, the processor grants to an I/O module the authority to read from or write to memory, so that the I/O-memory transfer can occur without tying up the processor. During such a transfer, the I/O module issues read or write commands to memory, relieving the processor of responsibility for the exchange. This operation, known as direct memory access (DMA), is examined Section 1.7.

## C.3 I/O COMMUNICATION TECHNIQUES

Three techniques are possible for I/O operations:

- Programmed I/O
- Interrupt-driven I/O
- Direct memory access (DMA)

### Programmed I/O

When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. In the case of programmed I/O, the I/O module performs the requested action and then sets the appropriate bits in the I/O status register but takes no further action to alert the processor. In particular, it does not interrupt the processor. Thus, after the I/O instruction is invoked, the processor must take some active role in determining when the I/O instruction is completed. For this purpose, the processor periodically

checks the status of the I/O module until it finds that the operation is complete.

With this technique, the processor is responsible for extracting data from main memory for output and storing data in main memory for input. I/O software is written in such a way that the processor executes instructions that give it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. Thus, the instruction set includes I/O instructions in the following categories:

- **Control:** Used to activate an external device and tell it what to do. For example, a magnetic-tape unit may be instructed to rewind or to move forward one record.
- **Status:** Used to test various status conditions associated with an I/O module and its peripherals.
- **Transfer:** Used to read and/or write data between processor registers and external devices.

Figure C.1a gives an example of the use of programmed I/O to read in a block of data from an external device (e.g., a record from tape) into memory. Data are read in one word (e.g., 16 bits) at a time. For each word that is read in, the processor must remain in a status-checking loop until it determines that the word is available in the I/O module's data register. This flowchart highlights the main disadvantage of this technique: It is a time-consuming process that keeps the processor busy needlessly.

## Interrupt-Driven I/O

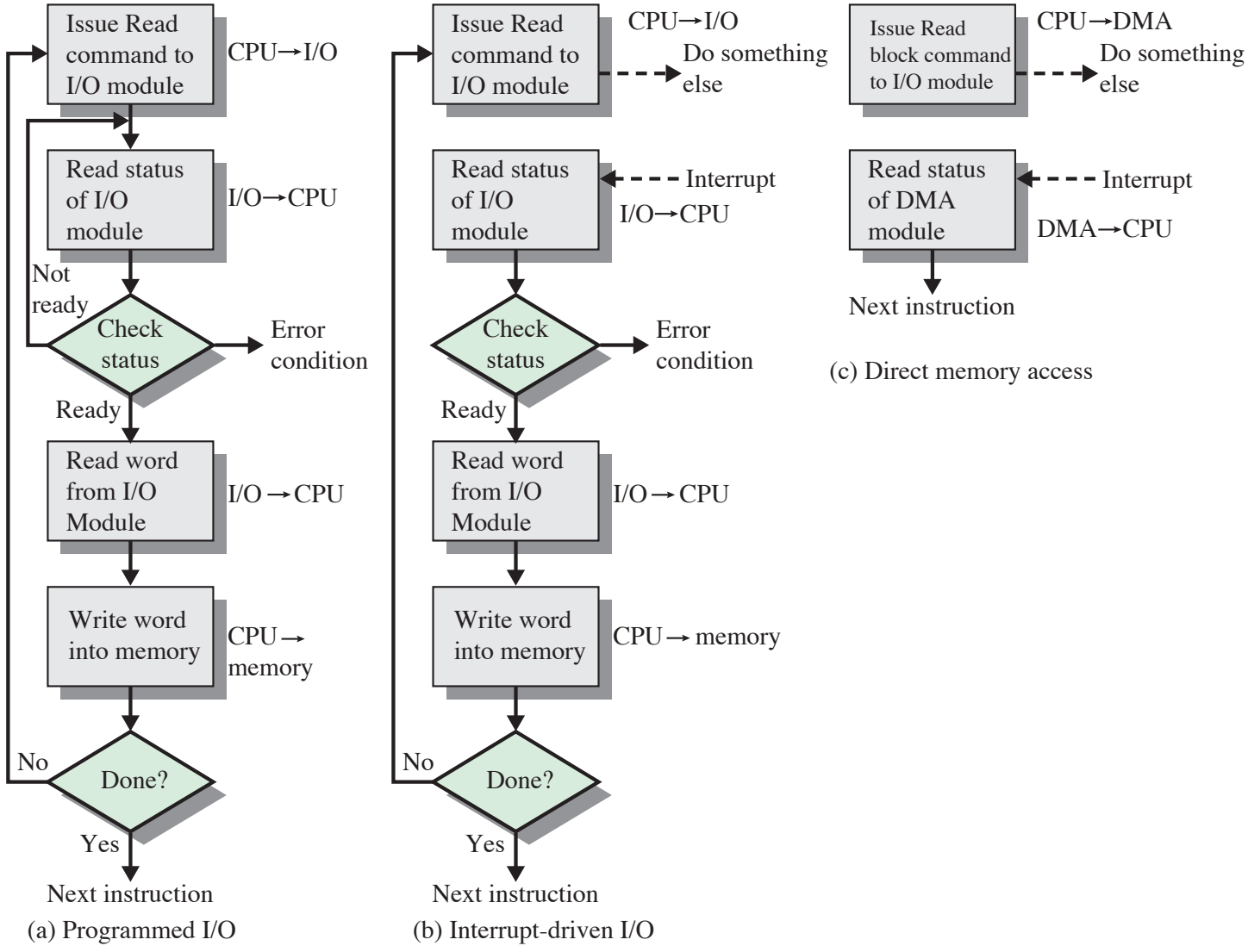



**Figure C.1 Three Techniques for Input of a Block of Data**

With programmed I/O, the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of more data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the performance level of the entire system is severely degraded.

An alternative is for the processor to issue an I/O command to a module and then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and then resumes its former processing.

Let us consider how this works, first from the point of view of the I/O module. For input, the I/O module receives a READ command from the processor. The I/O module then proceeds to read data in from an associated peripheral. Once the data are in the module's data register, the module signals an interrupt to the processor over a control line. The module then waits until its data are requested by the processor. When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

From the processor's point of view, the action for input is as follows. The processor issues a READ command. It then saves the context (e.g., program counter and processor registers) of the current program and goes off and does something else (e.g., the processor may be working on several different programs at the same time). At the end of each instruction cycle, the processor checks for interrupts (Figure 1.7). When the interrupt from the I/O module occurs, the processor saves the context of the program it is currently executing and begins to execute an interrupt-handling program that processes the interrupt. In this case, the processor reads the word of data from the I/O module and stores it in memory. It then restores the

context of the program that had issued the I/O command (or some other program) and resumes execution.

Figure C.1b shows the use of interrupt-driven I/O for reading in a block of data. Interrupt-driven I/O is more efficient than programmed I/O because it eliminates needless waiting. However, interrupt-driven I/O still consumes a lot of processor time, because every word of data that goes from memory to I/O module or from I/O module to memory must pass through the processor.

Almost invariably, there will be multiple I/O modules in a computer system, so mechanisms are needed to enable the processor to determine which device caused the interrupt and to decide, in the case of multiple interrupts, which one to handle first. In some systems, there are multiple interrupt lines, so that each I/O module signals on a different line. Each line will have a different priority. Alternatively, there can be a single interrupt line, but additional lines are used to hold a device address. Again, different devices are assigned different priorities.

## Direct Memory Access

Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor. Thus both of these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.
2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA). The DMA function can be performed by a separate module on the system bus or it can be incorporated into an I/O module. In either case, the technique works as follows. When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

- Whether a read or write is requested
- The address of the I/O device involved
- The starting location in memory to read data from or write data to
- The number of words to be read or written

The processor then continues with other work. It has delegated this I/O operation to the DMA module, and that module will take care of it. The DMA module transfers the entire block of data, one word at a time, directly to or from memory without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus the processor is involved only at the beginning and end of the transfer (Figure C.1c).

The DMA module needs to take control of the bus to transfer data to and from memory. Because of this competition for bus usage, there may be times when the processor needs the bus and must wait for the DMA module. Note that this is not an interrupt; the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle (the time it takes to transfer one word across the bus). The overall effect is to cause the processor to execute more slowly during a DMA transfer when processor access to the bus is required. Nevertheless, for a multiple-word I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

## C.4 HARDWARE PERFORMANCE ISSUES FOR MULTICORE

Microprocessor systems have experienced a steady, exponential increase in execution performance for decades. Figure C.2 shows that this increase is due partly to refinements in the organization of the processor on the chip, and partly to the increase in the clock frequency.

### Increase in Parallelism

The organizational changes in processor design have primarily been focused on increasing instruction-level parallelism, so that more work could be done in each clock cycle. These changes include, in chronological order (Figure C.3):

- **Pipelining:** Individual instructions are executed through a pipeline of stages so that while one instruction is executing in one stage of the pipeline, another instruction is executing in another stage of the pipeline.
- **Superscalar:** Multiple pipelines are constructed by replicating execution resources. This enables parallel execution of instructions in parallel pipelines, so long as hazards are avoided.
- **Simultaneous multithreading (SMT):** Register banks are replicated so that multiple threads can share the use of pipeline resources.

For each of these innovations, designers have over the years attempted to increase the performance of the system by adding complexity. In the case

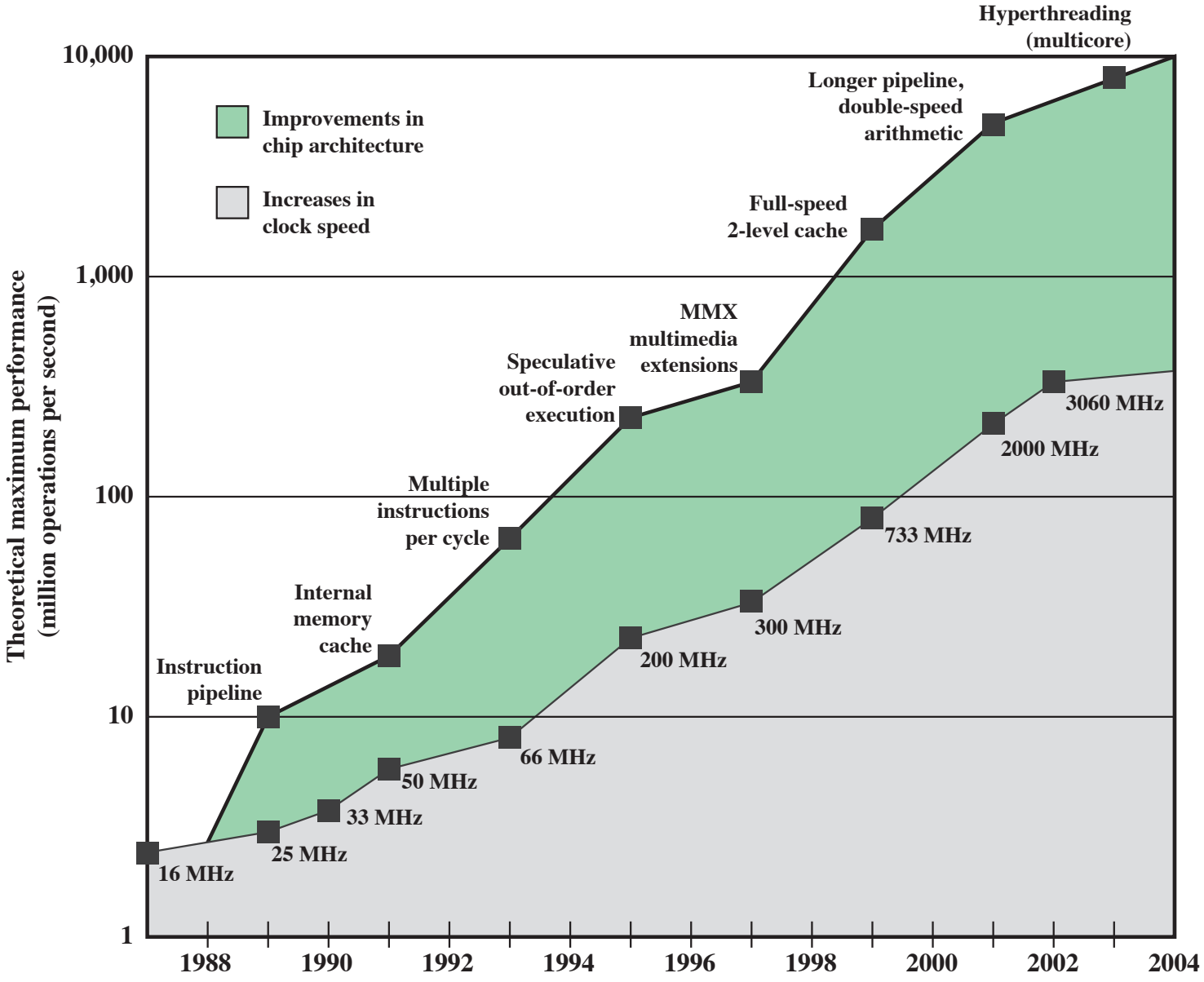
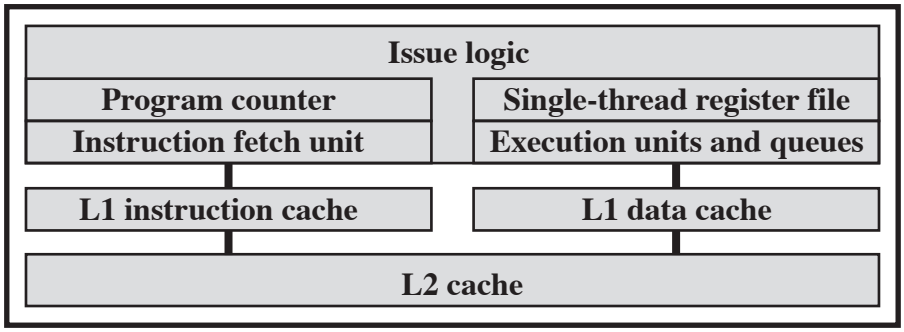
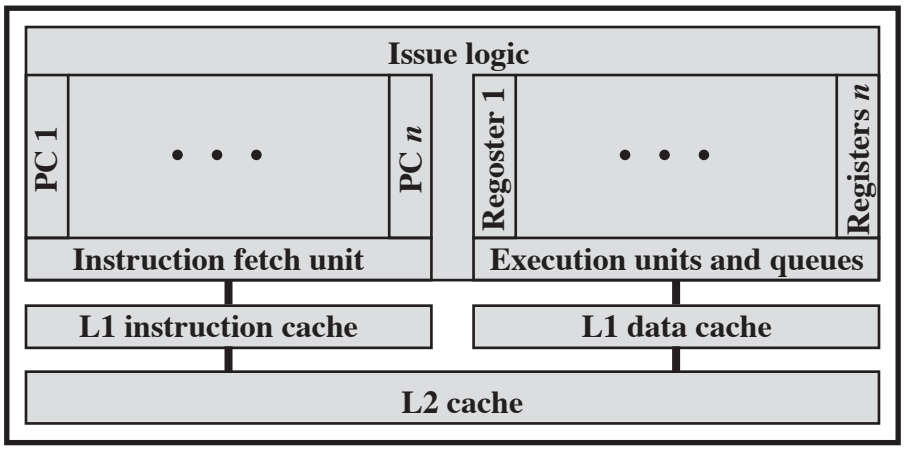


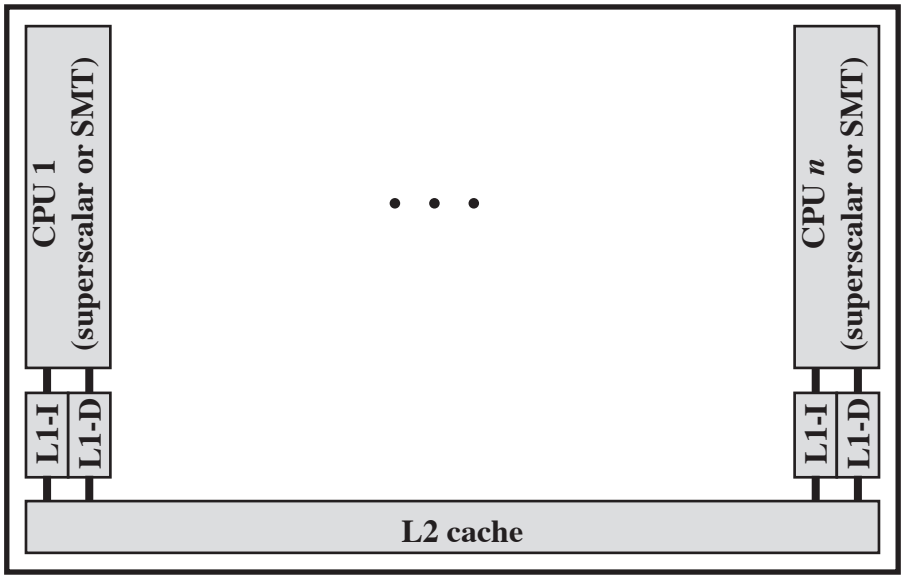

**Figure C.2 Intel Microprocessor Performance [GIBB04]**




(a) Superscalar




(b) Simultaneous multithreading




(c) Multicore

**Figure C.3 Alternative Chip Organizations**

of pipelining, simple three-stage pipelines were replaced by pipelines with five stages, and then many more stages, with some implementations having over a dozen stages. There is a practical limit to how far this trend can be taken, because with more stages, there is the need for more logic, more interconnections, and more control signals. With superscalar organization, performance increases can be achieved by increasing the number of parallel pipelines. Again, there are diminishing returns as the number of pipelines increases. More logic is required to manage hazards and to stage instruction resources. Eventually, a single thread of execution reaches the point where hazards and resource dependencies prevent the full use of the multiple pipelines available. This same point of diminishing returns is reached with SMT, as the complexity of managing multiple threads over a set of pipelines limits the number of threads and number of pipelines that can be effectively utilized.

Figure C.4, from [OLUK05], is instructive in this context. The upper graph shows the exponential increase in Intel processor performance over the years.[3] The middle graph is calculated by combining Intel's published SPEC CPU figures and processor clock frequencies to give a measure of the extent to which performance improvement is due to increased exploitation of instruction-level parallelism. There is a flat region in the late 1980s before parallelism was exploited extensively. This is followed by a steep rise as designers were able to increasingly exploit pipelining, superscalar techniques, and SMT. But, beginning about 2000, a new flat region of the curve appears, as the limits of effective exploitation of instruction-level parallelism are reached.

There is a related set of problems dealing with the design and fabrication of the computer chip. The increase in complexity to deal with all

---

[3] The data are based on published SPEC CPU figures from Intel, normalized across varying suites.

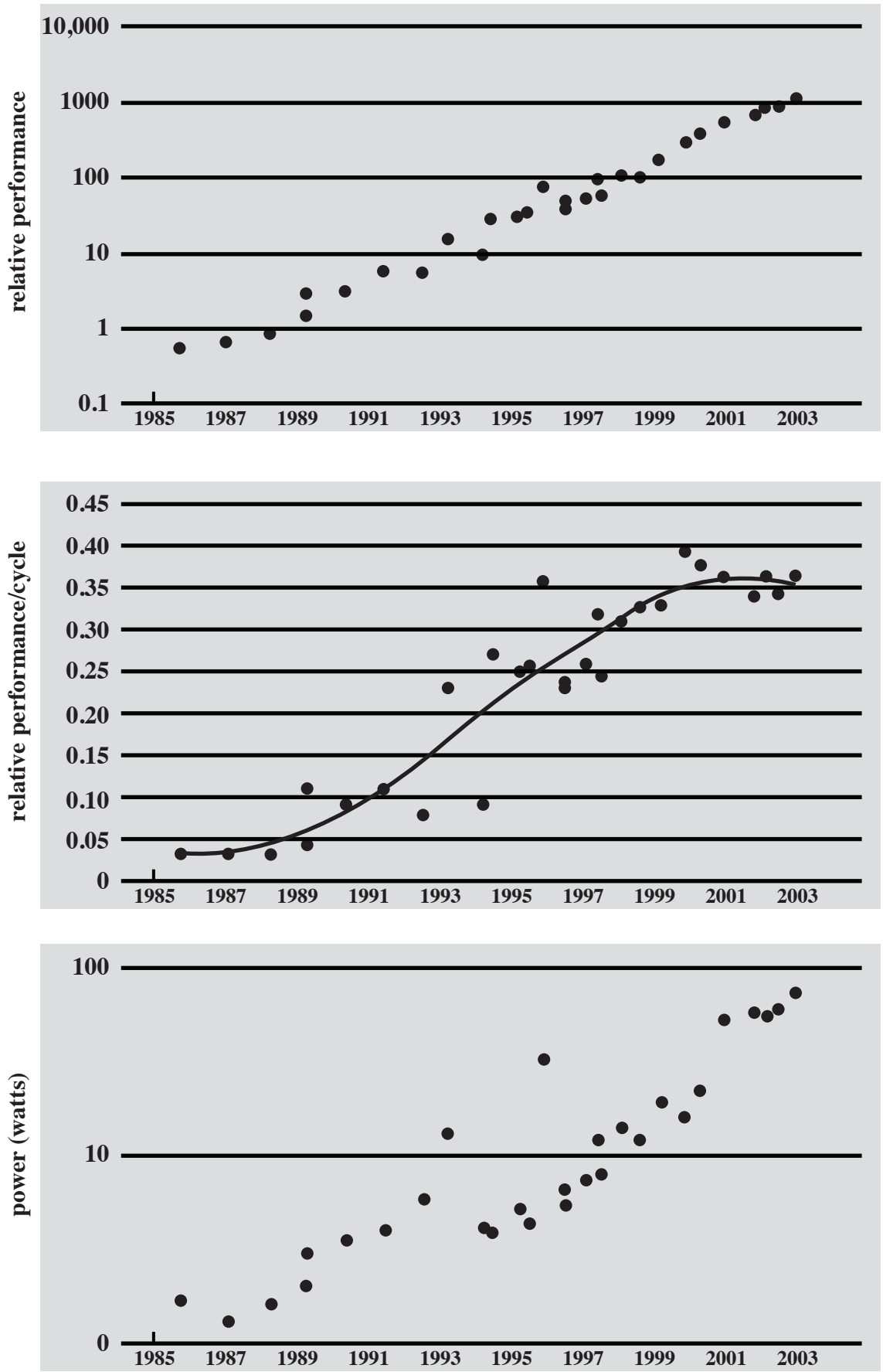



**Figure C.4 Some Intel Hardware Trends**

of the logical issues related to very long pipelines, multiple superscalar pipelines, and multiple SMT register banks means that increasing amounts of the chip area is occupied with coordinating and signal transfer logic. This increases the difficulty of designing, fabricating, and debugging the chips. The increasingly difficult engineering challenge related to processor logic is one of the reasons that an increasing fraction of the processor chip is devoted to the simpler memory logic. Power issues, discussed next, provide another reason.

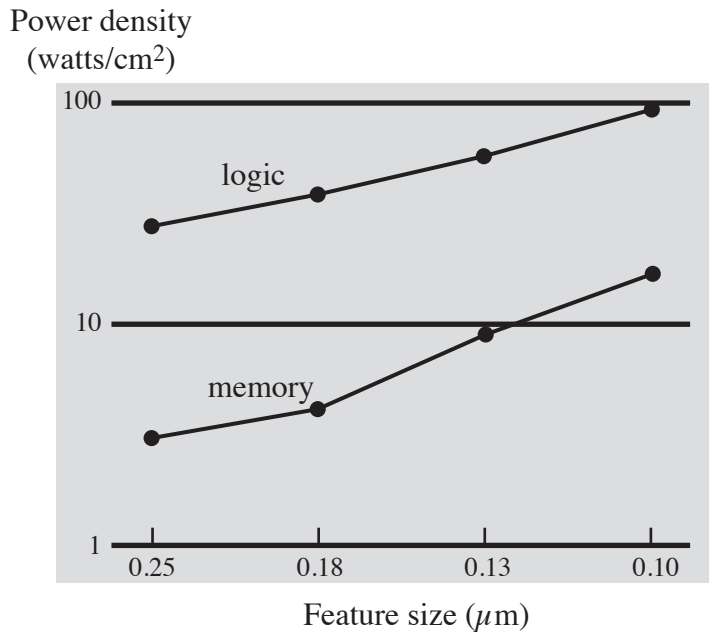
## Power Consumption

To maintain the trend of higher performance as the number of transistors per chip rise, designers have resorted to more elaborate processor designs (pipelining, superscalar, SMT) and to high clock frequencies. Unfortunately, power requirements have grown exponentially as chip density and clock frequency have risen. This is shown in the lowest graph in Figure C.4.

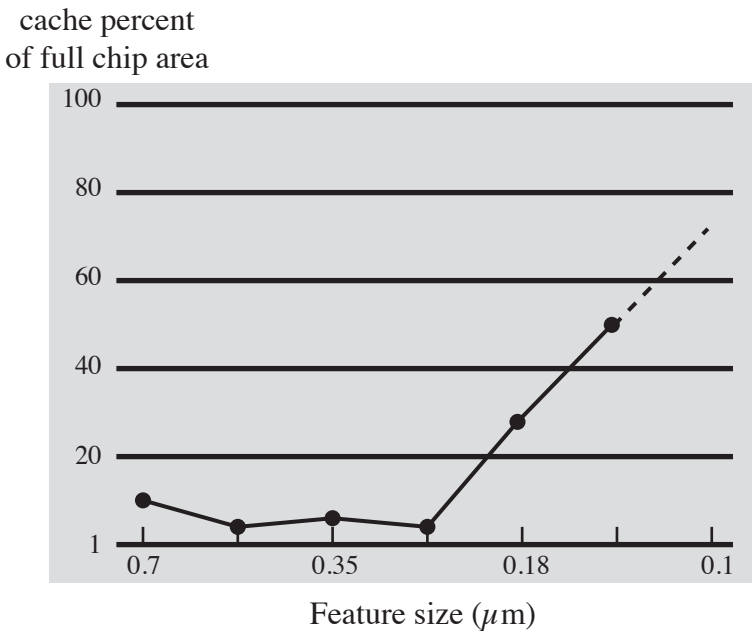
One way to control power density is to use more of the chip area for cache memory. Memory transistors are smaller and have a power density an order of magnitude lower than that of logic (see Figure C.5a). As Figure C.5b, from [BORK03], shows, the percentage of the chip area devoted to memory has grown to exceed 50% as the chip transistor density has increased.

Figure C.6, from [BORK07], shows where the power consumption trend is leading. By 2015, we can expect to see microprocessor chips with about 100 billion transistors on a 300 $mm^2$ die. Assuming about 50-60% of the chip area is devoted to memory, the chip will support cache memory of about 100 MB and leave over 1 billion transistors available for logic.

How to use all those logic transistors is a key design issue. As discussed earlier in this section, there are limits to the effective use of such techniques as superscalar and SMT. In general terms, the experience of recent decades




**(a) Power density**




**(b) Chip area**

# Figure C.5 Power and Memory Considerations

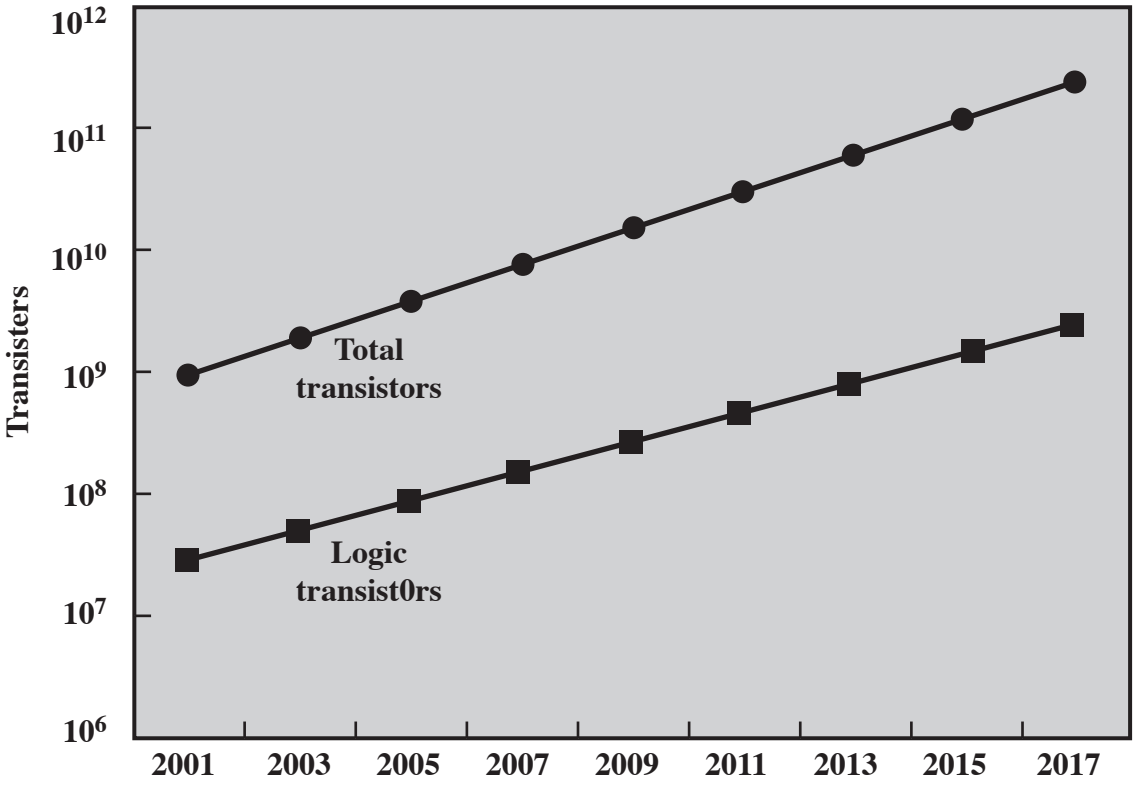



**Figure C.6 Chip Utilization of Transistors**

has been encapsulated in a rule of thumb known as **Pollack's rule** [POLL99], which states that performance increase is roughly proportional to square root of increase in complexity. In other words, if you double the logic in a processor core, then it delivers only 40% more performance. In principle, the use of multiple cores has the potential to provide near-linear performance improvement with the increase in the number of cores.

Power considerations provide another motive for moving toward a multicore organization. Because the chip has such a huge amount of cache memory, it becomes unlikely that any one thread of execution can effectively use all that memory. Even with SMT, you are multithreading in a relatively limited fashion and cannot therefore fully exploit a gigantic cache, whereas a number of relatively independent threads or processes has a greater opportunity to take full advantage of the cache memory.

## C.5 REFERENCES


**BORK03** Borkar, S. "Getting Gigascale Chips: Challenges and Opportunities in Continuing Moore's Law." *ACM Queue*, October 2003.

**BORK07** Borkar, S. "Thousand Core Chips—A Technology Perspective." *Proceedings, ACM/IEEE Design Automation Conference*, 2007.

**OLUK05** Olukotun, K., and Hammond, L. "The Future of Microprocessors." *ACM Queue*, September 2005.

**POLL99** Pollack, F. " New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (keynote address)." *Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture*, 1999.