Focusing on the Blurry Distinction between Microprocessors and Microcontrollers

J. T. Wunderlich
Elizabethtown College and Purdue University

Abstract

This paper compares microprocessors and microcontrollers in the context of teaching a sophomore level course where students have completed previous studies in digital circuits and programming. Discussing the similarities between these devices helps reinforce the understanding of the basic function of either device. Topics such as the "fetch-decode-execute" of an instruction cycle, or the memory-mapping of I/O provide good examples of similarities. Discussing the differences helps identify which device is most suitable for a given application. Topics such as mathematical computation capabilities or the ability to contain all needed functionality on a single chip provide good examples of differences. It is also important to study these devices in the context of historical trends since today's microcontrollers have evolved from past microprocessors. The microcontroller of the future could look more like today's microprocessors -- with a wider data bus, enhanced mathematical functionality, and numerous speed-up schemes. However, many of the unique features of microcontrollers are unlikely to be found in future microprocessors -- the separate memory for instructions and data is one example; the on-chip I/O control features such as analog-to-digital conversion and pulse-width-modulated outputs are other examples. The understanding of microprocessors and microcontrollers can also be enhanced by considering the differences between how programmers and engineers may view these devices. For example, a device could be selected for the programming power of the instruction-set, or for the simplicity of the instruction-set and minimization of additional circuitry.

I. Introduction

Teaching any subject in a historical context helps identify possible trends and focuses on fundamental principles that do not change over time. Although electronic computing began in the 1940's, the first CPU (central processing unit) packaged onto a single IC (integrated chip) was not available until 1971; it was the Intel 4004 4-bit microprocessor [1], [2]. From this simple beginning, many different microprocessor and microcontroller architectures have been developed. Today's typical Intel Pentium-based personal computer running Windows (i.e., "Wintel" machine) evolved from the Intel 4004 via the 8008, 8080, 8085, 286, 386, and 486 microprocessors. Similar advances by other companies led to *families* of microprocessors built around completely different instruction sets (e.g., Motorola 680xx family, Sparc family, etc.). The microprocessor typically studied in undergraduate courses is either an Intel or Motorola device. This is also true for microcontrollers (with the Intel 8051 and Motorola 68HC11 families [1], [3]); however, there are also some very simple 8-bit microcontrollers from other manufacturers that are gaining popularity -- with as few as 32 instructions and programmable with a personal computer [4]. Analyzing the simplest devices can help identify the basic features needed to make any microcontroller or microprocessor useful. Conversely, the study of *high-end* computer architectures can help identify the "cutting-edge" features needed to sustain the ever-increasing demand for more processing power in both microprocessors and microcontrollers.

Copyright © 1999, American Society for Engineering Education

II. Microprocessor and Microcontroller Similarities

Discussing the similarities between microcontrollers and microprocessors helps reinforce the understanding of the basic function of either device. A good way to visualize this is by approximating a "minimal-computer-architecture" common to both devices (see Fig. 1). This figure does not make a distinction between internal (on-chip) and external (off-chip) buses or memory, code space vs. data space, etc. -- but does attempt to get across the basic idea of how these devices function.

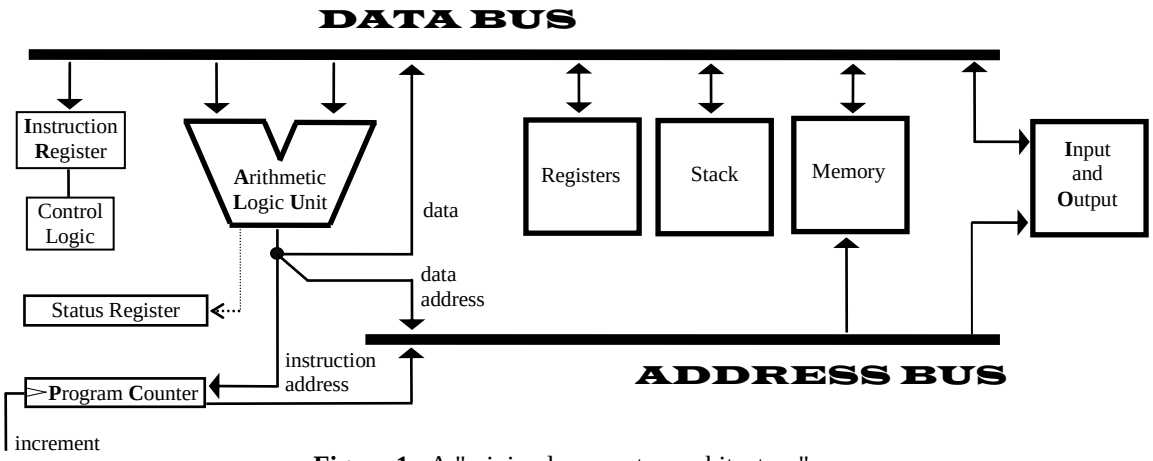

Figure 1. A "minimal-computer-architecture".

As shown in Fig. 1, each device contains:

- A program counter to address instructions to be fetched from memory.
- An instruction register to put the fetched instruction in.
- Control logic to create all routing signals after decoding the fetched instruction.
- An ALU for arithmetic and logical manipulation of data and addresses.
- Registers for storing intermediate results of calculations.
- A status register for status flags and condition codes.
- Memory for storing data and instructions.
- A stack for storing addresses (or processor status) for returning from program-calls (or interrupts).
- I/O which is addressed as memory (i.e., memory-mapped I/O).

Although the number of stages in an instruction cycle is device-dependent and can vary significantly, the fetch, decode, and execute stages are common to most microcontrollers and microprocessors. Whether instruction code is stored in internal (on-chip) or external (off-chip) ROM or RAM, it must be fetched via a data bus after being pointed to by a program counter which puts a memory address on an address bus. Both devices have instructions identified by op-codes which must be decoded during a decode phase; and each device has at least one execution phase where most control-actions are carried out. Additionally, both devices typically have a write-back phase for instructions that store ALU results into memory.

III. Microprocessor and Microcontroller Differences

Discussing the differences between microcontrollers and microprocessors helps identify which device is more suitable for a given application. A good way to examine differences is by working from the same "minimal-computer-architecture" shown above. One major difference is in how instructions and data are stored in memory, and where the memory is physically located. For typical microprocessors, all memory is located off-chip (with the exception of *some* caches) -- with a ROM used to permanently store instructions such as a bootstrap program to start-up the system, and RAM used to store data and all other instructions (i.e., "Von Neumann" architecture). For many microcontrollers, there is ROM and RAM on the chip, *address-space* for additional ROM and RAM off-chip, and registers and a stack that can be addressed as internal RAM. Also, instructions and data are stored separately (i.e., "Harvard" architecture); the ROM is used to contain all instructions upon completion of code development, and the RAM is used to store data. However, during code development the RAM can

Copyright © 1999, American Society for Engineering Education

be used to store both code and data. This architecture is well suited for embedded applications where the code is fixed (i.e., burned into ROM). Having on-chip RAM also helps allow the microcontroller to be embedded as a single-chip computer.

In microprocessors, on-chip and off-chip caches (and often separate caches for data and instructions) are used to speed-up processing. These memories are made from fast static-RAM (i.e., faster, but more expensive and with less transistor-density than dynamic-RAM), and contain *most-recently-used* data and instructions (which are statistically more likely to be needed).

Both microcontrollers and microprocessors have a program counter to address instructions to be fetched from memory. However, the calculation of instruction addresses can be significantly more complex in microprocessors; with the target-addresses of branch instructions being prefetched using branch-prediction strategies; and with several levels of address translation required to get *real* RAM addresses from *virtual* addresses when the address space is not directly mapped to the actual available RAM.

Both devices have an instruction register to receive fetched instructions, and circuitry to decode instruction op-codes, however the control logic to create all routing signals is more complex for microprocessors which have more powerful instruction sets -- with fancier addressing modes and many instructions. The only exception to this is RISC microprocessors (i.e., Reduced Instruction-Set Computer) where intentionally simple instruction-sets and addressing modes are developed as part of the overall speed-up scheme for the processor [5]. Although microcontrollers have simple instruction-sets, they do not typically have other features needed to classify them as RISC devices (e.g., fixed instruction-length formats). They do however have on-chip RAM and several general-purpose register banks which allow faster access of data than off-chip memory-accesses -- and this has the same effect as the large register-sets found on RISC chips. Both devices have a stack for storing addresses (or processor status) for returning from program-calls (or interrupts), however microprocessors often have two stacks: a general-purpose user-stack, and a system-stack that requires *privileged* instructions to access [6].

The evolution of microprocessors (as well as more complex high-performance machines) has led to many advances in computer architecture to speed-up processing; this has included much more than increasing processor clock speed. The time to execute a program can be represented by:

$$T = \overline{CPI} * (I_c) * \tau \qquad (1)$$

where τ is the clock period in seconds per cycle (i.e., 1/*frequency*), I_c is the number of machine instructions in a given code segment, and $\overline{CPI}$ (cycles per instruction) is the average time to fetch, decode, execute, and store results for each instruction [5]. There are many strategies to decrease $\overline{CPI}$; for example, processing several instructions simultaneously (i.e., superscalar), or moving data directly between I/O and memory (i.e., Direct Memory Access). Hardware to anticipate and take "*pre*-actions" has been a design concept for many years. This not only includes prefetching data and instructions in caches, but also prefetching branch-target addresses using *Branch History Tables*, or *caching* virtual address translations using *Translation Lookaside Buffers*. Other speed-up techniques include re-ordering and optimizing instruction streams as they come into the CPU (i.e., out-of-order execution), or overlapping the individual instruction-cycle phases of many instructions (i.e., super-pipelined). Many of these advances will eventually work their way into microcontroller architectures, however features designed to handle large address spaces are less likely to be needed for microcontroller applications.

Although both devices have an ALU for integer arithmetic and logical manipulation of data, microprocessors are much better suited for "number-crunching", and usually have a *wider* data bus and *larger* general-purpose registers to accommodate this. Microcontrollers are typically limited to 8-bit or 16-bit number representations (even though 32-bit microcontrollers are available); whereas microprocessors usually allow 32-bit representations, and contain additional floating-point hardware to allow arithmetic using much larger number ranges (and therefore much greater precision). Table 1 shows the available number range for different integer number representations.

Copyright © 1999, American Society for Engineering Education

Table 1. Number range for different integer number representations.

8-bit unsigned:	0 to (2^8)-1	=	0 to 255
8-bit signed:	-(2^8)/2 to ((2^8)/2)-1	=	-128 to 127
16-bit unsigned:	0 to (2^16)-1	=	0 to 65,535
16-bit signed:	-(2^16)/2 to ((2^16)/2)-1	=	-32,768 to 32,767
32-bit unsigned:	0 to (2^32)-1	=	0 to 4,294,967,295
32-bit signed:	-(2^32)/2 to ((2^32)/2)-1	=	-2,147,483,648 to 2,147,483,647
n-bit unsigned:	0 to (2^n)-1		
n-bit signed:	-(2^n)/2 to ((2^n)/2)-1		

One might argue that 8-bit microcontrollers are capable of manipulating large number ranges by simply parsing and manipulating large numbers using 8-bit quantities; however, as shown in Fig.'s 2 and 3, even a simple operation like decrementing a 16-bit number until it equals another 16-bit number can be much more tedious using 8-bit arithmetic. Even if the 8051 microcontroller programming effort in Fig. 3 was avoided by programming in a high-level language (e.g., C), the resulting machine code after compilation would still require many more bytes than if 16-bit registers where available (i.e., 22 vs. 8 for these examples). The limited 8051 instruction-set also contributes to the many lines of code needed in Fig. 3; the accumulator must be used to receive all results from the ALU, therefore forcing intermediate results to be moved in and out of the accumulator (i.e., at lines #8 and #11); whereas a typical microprocessor will have many register-to-register instructions, allowing intermediate results of computations to be left in several register locations (see MC68000 code in Fig. 2)[6]. Although large program size is not usually a problem for microprocessor systems, it can be a problem for microcontrollers which can have limited space for code (i.e., in ROM) -- especially if all of the code is to fit in the on-chip ROM.

Figure 2. Example MC68000 microprocessor program using 16-bit arithmetic to do a 16-bit task; Decrement the 16-bits in general-purpose data register D0 until it reaches the 16-bit number in general-purpose data register D2.

```
                                                                          # OF      # OF
LINE                                                                      BYTES   CYCLES
01        check:  CMP.W D0, D2      ; compare D0 and D2, set appropriate condition flag     2       4
02                DBE D0, check     ; decrement, and jump to " check "  until D0 and D2 equal  4    10 to 12
03        done:   NOP               ; program finished                                     2       4
                                                                                          =====
                                                                              TOTAL =     8
```

Figure 3. Example 8051 microcontroller program using 8-bit arithmetic to do a 16-bit task; Decrement the 8-bit general-purpose registers R1 and R0 as one concatenated 16-bit number until it reaches the 16-bit number made by concatenating the contents of the 8-bit general-purpose registers R3 and R2.

```
                                                                          # OF      # OF
LINE                                                                      BYTES   CYCLES
00        check:  MOV A, R0                 ;put low-order byte in accumulator                1       1
01                CJNE A, 02h, dcrmnt ;     conditional jump to "dcrmnt" if not equal to R2 contents  3   2
02                MOV A, R1                 ;put high-order byte in accumulator               1       1
03                CJNE A, 03h, dcrmnt;      ;conditional jump to " dcrmnt " if not equal to R3 contents  3  2
04                SJMP done                 ;countdown finished, jump to "done"               2       2
05        dcrmnt: MOV A, R0                 ;put low-order byte in accumulator                1       1
06                CLR C                     ;must clear carry flag since used in subtraction  1       1
07                SUBB A, #01h              ;decrement  (and possibly set borrow)             2       1
08                MOV R0 ,A                 ;temporarily store new high-order byte in R0       1       1
09                MOV A, R1                 ;put high-order byte in accumulator               1       1
10                SUBB A, #00h              ;subtract borrow (i.e., carry bit is set if borrow at line #07)  2  1
11                MOV R1 ,A                 ;temporarily store new high-order byte in  R1     1       1
12                SJMP check                ;jump to "check "                                 2       2
13        done:   NOP                       ;program finished                                 1       1
                                                                                          =====
                                                                              TOTAL =     22
```


Copyright © 1999, American Society for Engineering Education

One strength of microcontrollers is their on-chip RAM which allows faster memory access and therefore fewer cycles per instruction. This is illustrated in Fig.'s 4 and 5. Although the MC68000 microprocessor code of Fig. 4 requires less bytes, it must access the off-chip RAM for reading and writing the initial and final *count*; this is significantly slower than manipulating on-chip RAM. However, the overall speed of the MC68000 microprocessor code in Fig. 4 would be as fast as the 8051 microcontroller code of Fig. 5 if the difference between the initial *count* and the desired *count* was large enough (assuming equal clock speeds). The above discussion can be easily extended to a comparison of 16-bit and 32-bit devices (i.e., when doing 32-bit arithmetic).

Figure 4. Example MC68000 microprocessor program using 16-bit arithmetic to do a 16-bit task; Decrement the 16-bits at RAM location 2000h until it reaches the 16-bit number in general-purpose data register D2; then store count back into memory.

LINE				# OF BYTES	# OF CYCLES
00		MOVE.W $2000, DO	; copy original count into register D0 from RAM (off-chip)	4	12
01	check:	CMP.W D0, D2	; compare D0 and D2, set appropriate condition flag	2	4
02		DBE D0, check	; decrement, and jump to " check " until D0 and D2 equal	4	10 to 12
03		MOVE.W D0, $2000	; write count to RAM (off-chip) from D0	4	12
04	done:	NOP	; program finished	2	4
				=====	
			TOTAL =	16	

Figure 5. Example 8051 microcontroller program using 8-bit arithmetic to do a 16-bit task; Decrement the 8-bit contents of internal RAM addresses 21h and 20h as one concatenated 16-bit number until it reaches the 16-bit number made by concatenating the contents of the 8-bit general-purpose registers R3 and R2 [2].

LINE				# OF BYTES	# OF CYCLES
00	check:	MOV A, 20h	;get low-order byte from on-chip RAM	2	1
01		CJNE A, 02h, dcrmn t	;conditional jump to "dcrmnt" if not equal to R2 contents	3	2
02		MOV A, 21h	;get high-order byte from on-chip RAM	2	1
03		CJNE A, 03h, dcrmnt	;conditional jump to " dcrmnt " if not equal to R3 contents	3	2
04		SJMP done	;countdown finished, jump to "done"	2	2
05	dcrmnt:	MOV A, 20h	;get low-order byte from on-chip RAM for decrementing	2	1
06		CLR C	;must clear carry flag since it is used as a borrow	1	1
07		SUBB A, #01h	;decrement (and possibly set borrow)	2	1
08		MOV 20h ,A	;store new high-order byte in on-chip RAM	2	1
09		MOV A, 21h	;get high-order byte from on-chip RAM for decrementing	2	1
10		SUBB A, #00h	;subtract borrow (i.e., carry bit is set if borrow at line #07)	2	1
11		MOV 21h ,A	;store new low-order byte in on-chip RAM	2	1
12		SJMP check	;jump to "check "	2	2
13	done:	NOP	;program finished	1	1
				=====	
			TOTAL =	28	

Probably the most defining characteristic of microcontrollers is the on-chip circuitry for interfacing with external devices. This includes watchdog timers, analog-to-digital and digital-to-analog converters (ADC's and DAC's), pulse-width-modulated (PWM) outputs for driving motors, and many counters for timing and control. These circuits are not typically found on microprocessors.

The microcontroller is best suited as an embedded single-chip computer for controlling peripheral devices, whereas the microprocessor is best suited for relatively high-speed general-purpose computing, high-precision math-intensive applications (e.g., multimedia, scientific simulations, etc.), or programming which makes use of large address spaces. However, as the number of features that can fit on a single chip increases, the microcontroller and microprocessor of the future could be packaged as a single device -- a single-chip computer with all the advantages of both, where the engineer or programmer could select which features to use. Although this might seem wasteful, mass production could make these devices inexpensive enough to justify this strategy.

Copyright © 1999, American Society for Engineering Education

IV. Different Perspectives

The understanding of microprocessors and microcontrollers can also be enhanced by considering the differences between how programmers and engineers may view these devices. The complex instruction-set and addressing modes of most microprocessors might be considered a big advantage by systems-level programmers who are willing (and able) to make use of these features -- this bias may even outweigh the on-chip features of microcontrollers. For example, if a microprocessor or microcontroller needed to be chosen for an application requiring analog numbers to be read into the device, then manipulated using 16-bit arithmetic, then displayed on analog meters, the programmer might decide that the code could be most effectively written for a 16-bit microprocessor and therefore not choose a 16-bit microcontroller with built-in analog-to-digital (ADC) and digital-to-analog (DAC) converters, on-chip ROM that might fit all the code, and on-chip RAM. An engineer however might choose a 16-bit microcontroller because of the simpler instruction-set (even though more lines of program code might be required), or because the on-chip features eliminate the need to design board-level circuits to handle analog conversions and communication between the CPU and memory.

V. Laboratory Experiments

Most courses in microprocessors or microcontrollers can benefit from laboratory experiments. Appendix A. lists the laboratory projects used for a Purdue University sophomore microcontroller course [8]. These labs use an 8051 microcontroller-based test computer (the PU-552), and monitor program developed in the Purdue University Electrical and Computer Engineering Technology Department [2], [8]. If a microprocessor-based development system is used, some of the device-control type labs can be replaced with, for example, floating-point arithmetic exercises. The programming language used for the labs in Appendix A. is mostly 8051 Assembly -- however C programming is also required for several labs. Students can also benefit from comparing code written in both C and Assembly for the same task.

The typical undergraduate course in microprocessors or microcontrollers requires prerequisite studies in digital circuits. The lab project shown in Fig. 6 can be used in a prerequisite digital-design course to introduce the control and flow of data in a microprocessor or microcontroller.

VI. Conclusions

A good way to learn the differences between microprocessors and microcontrollers is to work from a "minimal-computer-architecture" common to both. This reinforces the understanding of basic computer architecture fundamentals, and isolates device differences so they can be easily identified. Discussing differences also helps identify which device is most suitable for a given application. Topics such as mathematical capabilities or the ability to contain all needed functionality on a single chip provide important examples of differences.

Teaching any subject in a historical context helps identify trends, and focuses on fundamental principles that do not change over time. Since the beginning of the packaged CPU in 1971, the evolution of microprocessors has led to many advances in computer architecture; many of which will eventually work their way into microcontrollers. Future microcontrollers could look like today's microprocessors -- with enhanced mathematical functionality, and numerous instruction speed-up schemes; however, some of the *peripheral-control* features of microcontrollers are unlikely to be found in future microprocessors. It may also become possible that as the number of features that can fit on a single chip increases, future microcontrollers and microprocessors could be packaged as a single device -- a single-chip computer with all the advantages of both, where the engineer or programmer could select which features to use.

The understanding of microprocessors and microcontrollers can also be enhanced by considering the differences between how programmers and engineers may view these devices. For example, a device could be selected for the programming power of the instruction-set, or for the simplicity of the instruction-set and minimization of additional circuitry.

Copyright © 1999, American Society for Engineering Education

Appendix A.

Laboratory projects used for a Purdue University Electrical and Computer Engineering Technology microcontroller course [8]:

1. Equipment orientation and I/O with C
2. Monitor program and program execution
3. Bus cycle timing analysis
4. Memory and I/O expansion
5. Move instructions and hand assembly
6. Branching and math instructions
7. Timing loops
8. Parallel I/O and C program development
9. Keypad and 7-segment LED displays
10. Serial I/O
11. ADC's and DAC's
12. Stepper motors and Digitalkers
13. Individual projects
14. Lab practical exam

Appendix B.

Figure 6 shows a digital-design course laboratory project to introduce the control and flow of data in a microprocessor or microcontroller. The counters are analogous to timers in a microcontroller or general-purpose registers used as counters in a microprocessor. The select line to the multiplexer can represent a *control-logic* signal generated after decoding the op-code; the comparator can represent a simplified arithmetic logic unit (ALU); and the L.E.D. circuits controlled by the comparator output can represent the contents of a status register. A variation of this lab can be made by replacing the comparator with a 2-bit parallel adder.

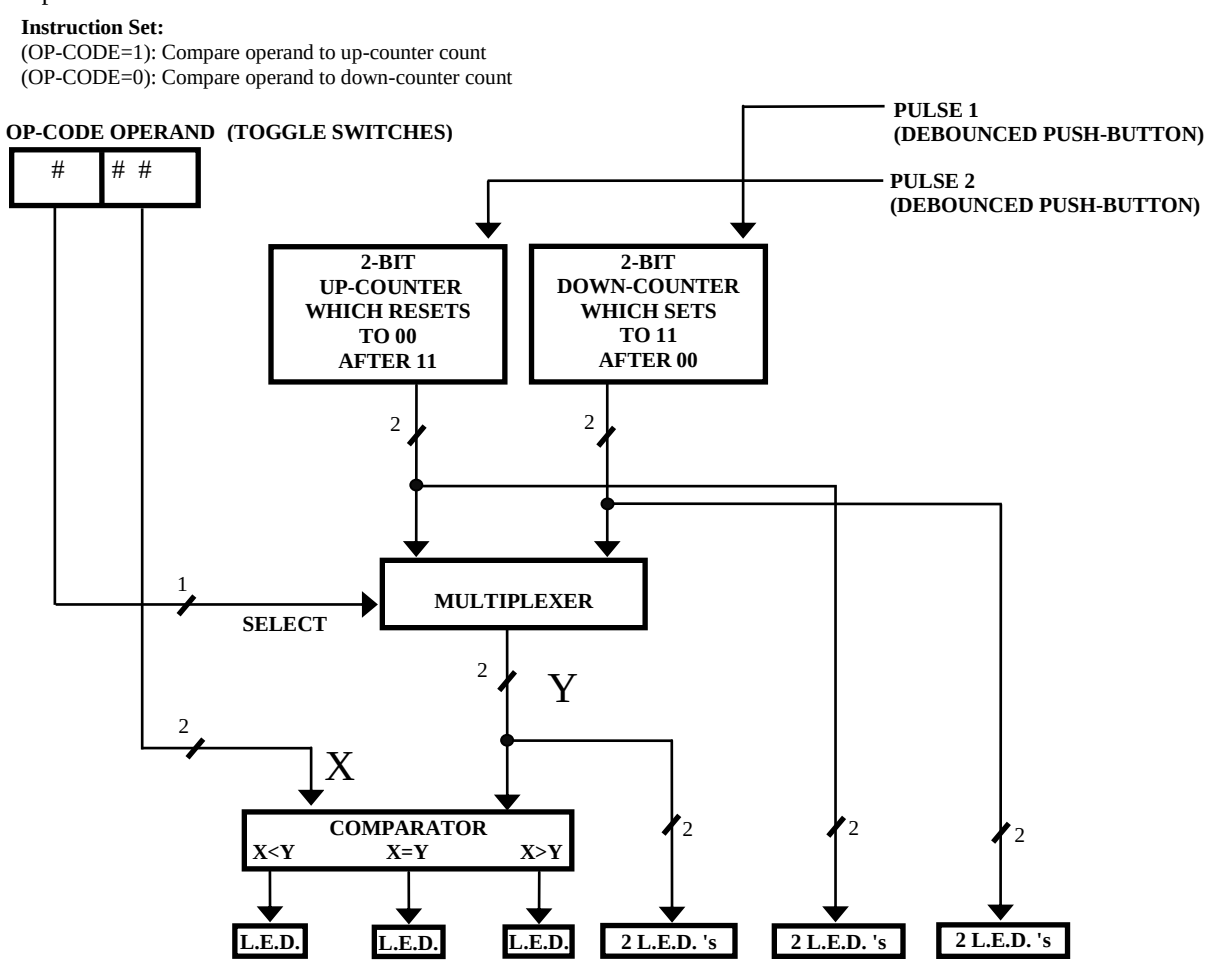

Figure 6. A simple digital-design course laboratory project to introduce the control and flow of data in a microprocessor or microcontroller.

Copyright © 1999, American Society for Engineering Education

References

[1] K. J. Ayala, *"The 8051 Microcontroller"*, 2nd ed., Minneapolis, MN: West Publishing, 1997.

[2] R. H. Barnett, *"The 8051 Family of Microcontrollers"*, Englewood Cliffs, NJ: Prentice Hall, 1995.

[3] M. Kheir, *"The M68HC11 Microcontroller"*, Upper Saddle River, NJ: Prentice Hall, 1997.

[4] J. B. Peatman, *"Design with PIC Microcontrollers"*, Upper Saddle River, NJ: Prentice Hall, 1998.

[5] K. Hwang, *"Advanced Computer Architecture: Parallelism, Scalability, Programmability"*, McGraw-Hill, 1993.

[6] T. L. Harman and B. Lawson, *"The Motorola 68000 Microprocessor Family"*, Englewood Cliffs, NJ: Prentice Hall, 1985.

[7] G. G. Langdon, *" Computer Design"*, San Jose, CA: Computeach Press, 1982.

[8] N. S. Widmer, *"Introduction to Microprocessors Laboratory Manual"*, West Lafayette, IN: Learning Systems, 1998.

Copyright © 1999, American Society for Engineering Education