Embedded Robotics

Mobile Robot Design and Applications with Embedded Systems

Bearbeitet von
Thomas Bräunl

Neuausgabe 2008. Taschenbuch. xiv, 546 S. Paperback
ISBN 978 3 540 70533 8
Format (B x L): 17 x 24,4 cm
Gewicht: 1940 g

2

Central Processing Unit

The CPU (central processing unit) is the heart of every embedded system and every personal computer. It comprises the ALU (arithmetic logic unit), responsible for the number crunching, and the CU (control unit), responsible for instruction sequencing and branching. Modern microprocessors and microcontrollers provide on a single chip the CPU and a varying degree of additional components, such as counters, timing coprocessors, watchdogs, SRAM (static RAM), and Flash-ROM (electrically erasable ROM).

Hardware can be described on several different levels, from low-level transistor-level to high-level hardware description languages (HDLs). The so-called register-transfer level is somewhat in-between, describing CPU components and their interaction on a relatively high level. We will use this level in this chapter to introduce gradually more complex components, which we will then use to construct a complete CPU. With the simulation system *Retro* [Chansavat Bräunl 1999], [Bräunl 2000], we will be able to actually program, run, and test our CPUs.

One of the best analogies for a CPU, I believe, is a mechanical clockwork (Figure 2.1). A large number of components interact with each other, following the rhythm of one central oscillator, where each part has to move exactly at the right time.

Figure 2.1: Working like clockwork

2.1 Logic Gates

On the lowest level of digital logic, we have logic gates AND, OR, and NOT (Figure 2.2). The functionality of each of these three basic gates can be fully described by a truth table (Table 2.1), which defines the logic output value for every possible combination of logic input values. Each logic component has a certain delay time (time it takes from a change of input until the corrected output is being produced), which limits its maximum operating frequency.

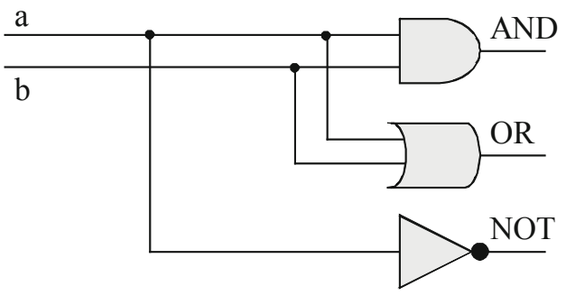


Figure 2.2: AND, OR, NOT gates

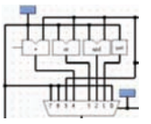
Input a, b	Output a AND b	Output a OR b	Output NOT a
0, 0	0	0	1
0, 1	0	1	1
1, 0	0	1	0
1, 1	1	1	0

Table 2.1: Truth table

Gates are built by using electronically activated switches. These are transistors in today's technology, while relays and vacuum tubes have been used in the past. However, for the understanding of the material in this chapter, we do not need to know any further low-level details.

The layer of abstraction above gate-level is formed by so-called combinatorial logic circuits. These do not have any timing components, and so everything can be explained as a combination of AND, OR, NOT gates.

In the following we will denote negated signals with an apostrophe (e.g. *a'* for *NOT a*) in text, and as a dot in a gate's input or output in diagrams (see Figure 2.3).



2.1.1 Encoder and Decoder

A decoder can be seen as a translator device of a given binary input number. A decoder with *n* input lines has 2^n output lines. Only the output line corresponding to the binary value of the input line will be set to "1", all other output lines will be set to "0". This can be described by the formula:

$$Y_i = \begin{cases} 1 & \text{if } i = X \\ 0 & \text{else} \end{cases}$$

Only the output line matching the binary input pattern is set to "1".

So if e.g. $n = 4$ and input X is a binary 2, meaning X_1=1 and X_0=0, then output line Y_2 will be "1", while Y_0, Y_1, and Y_3 will be "0".

Figure 2.3 shows a simple decoder example with two input lines and consequently four output lines. Its implementation with combinatorial logic requires four AND gates and four NOT gates. Decoders are being used as building blocks for memory modules (ROM and RAM) as well as for multiplexers and demultiplexers.

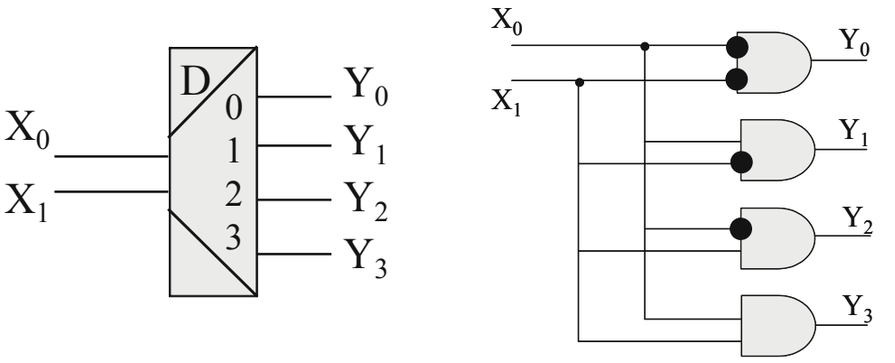


Figure 2.3: Decoder symbol and implementation

Encoders perform the opposite function of a decoder. They work under the assumption that only a single one of their input lines is active at any time. Their output lines will then represent the input line number as a binary number. Consequently, encoders with *n* output lines have 2^n input lines. Figure 2.4 shows the implementation of an encoder using only two OR gates. Note that X_0 is not connected to anything, as the output lines will default to zero if none of the other X lines are active. Figure 2.5 shows the interaction between an encoder and a decoder unit, reconstructing the original signal lines.

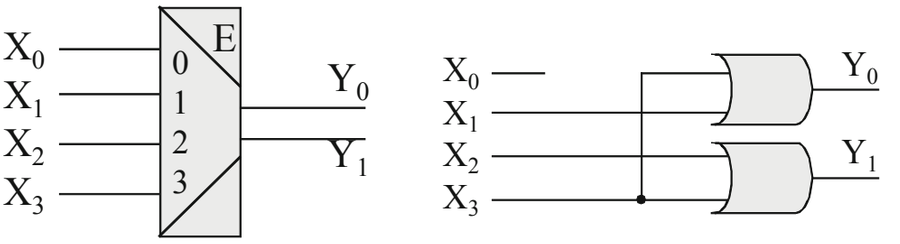


Figure 2.4: Encoder symbol and implementation

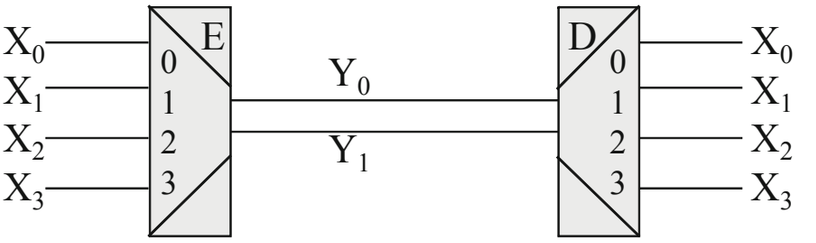


Figure 2.5: Encoder and Decoder

2.1.2 Multiplexer and Demultiplexer

The next level of abstraction are multiplexers and demultiplexers. A multiplexer routes exactly one of its inputs (X_1, ..., X_n) through to its output Y, depending on the selection lines S. Each input X_i and output Y have the same width (number of lines), and so they can either be a single line as in Figure 2.6 or can all be e.g. 8-bit wide.

The width (number of lines) of selection line S depends on the number of multiplexer inputs *n*, which is always a power of 2:

Number of inputs $n = 2^k$, with *k* being the width of S.

In the example in Figure 2.6, we have only two inputs, and so we need only a single selection line to distinguish between them. In this simple case, we can write the logic equation for a multiplexer as:

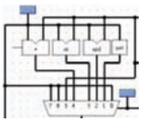
$$Y := S \cdot X_1 + S' \cdot X_0$$

The equivalence circuit built from AND, OR, and NOT gates is shown on the right-hand-side of Figure 2.6.

When building a larger multiplexer, such as the four-way multiplexer in Figure 2.7, using a decoder circuit makes the implementation a lot easier (Figure 2.7, right). For each case, the input position matching the selection lines is routed through, which can be written in short as:

$$Y := X_S$$

A demultiplexer has the opposite functionality to a multiplexer. Here we connect a single input X to one of several outputs $Y_1..Y_n$, depending on the



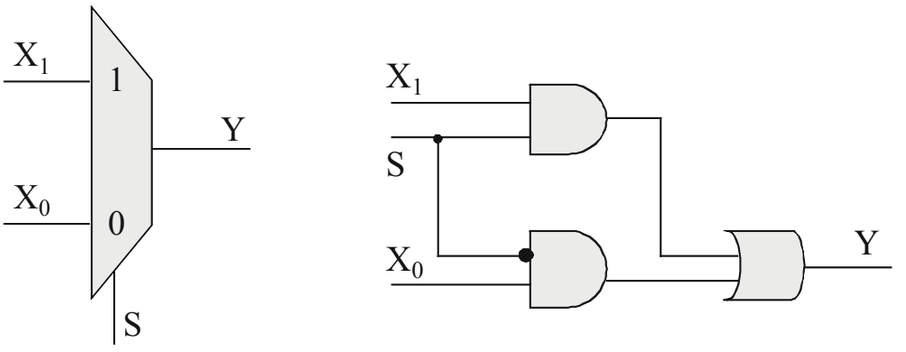


Figure 2.6: Multiplexer 2-way and implementation

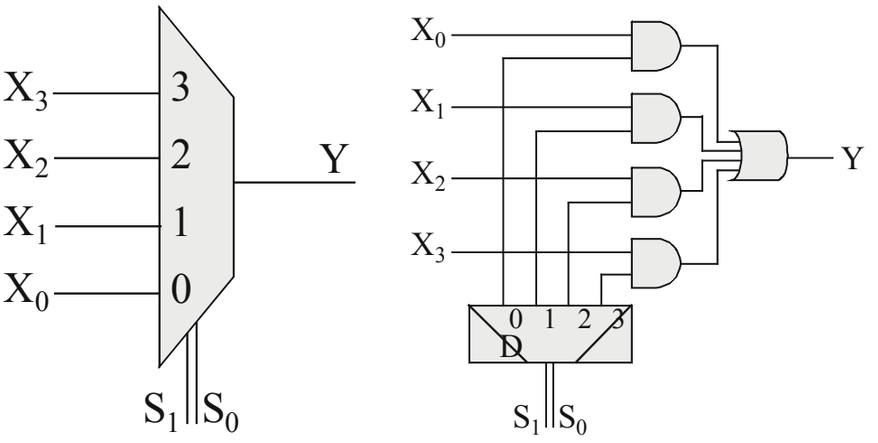


Figure 2.7: Multiplexer 4-way and implementation

status of the selection line S. In fact, if multiplexers and demultiplexers were built like a mechanical pipe system, they would be the same thing – just turning it around would make a multiplexer a demultiplexer and vice versa. Unfortunately, in the electronics world, it is not so easy to exchange inputs and outputs. Most electronic circuits have a "direction", as it becomes clear from the demultiplexer's equivalence circuit made out of AND and NOT gates in Figures 2.8 and 2.9.

The logic formula for a general demultiplexer is very similar to a decoder, however, remember that input X and outputs Y_i can be wider than a single line:

$$Y_i = \begin{cases} X & \text{if } i = S \\ 0 & \text{else} \end{cases}$$

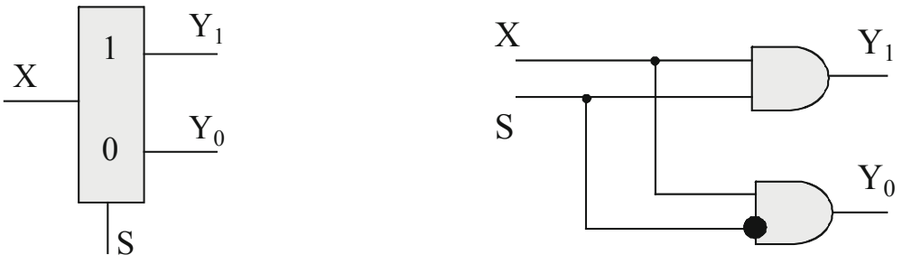


Figure 2.8: Demultiplexer 2-way and implementation

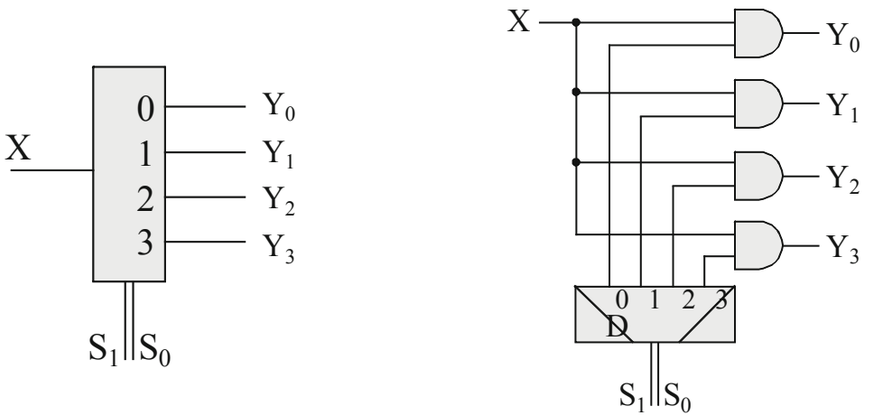


Figure 2.9: Demultiplexer 4-way and implementation

2.1.3 Adder

The adder is a standard textbook example, and so we can be very brief about it. The first step is building a half-adder that can add 2-bit input (X, Y) and produce 1-bit output plus a carry bit. It can be constructed by using an XOR and an AND gate (Figure 2.10).

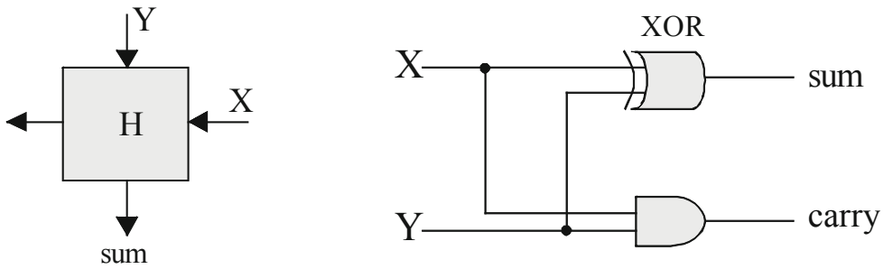
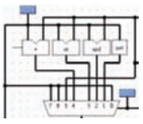


Figure 2.10: Half-Adder symbol (2-bit) and implementation



Two half-adders and an OR gate are being used to build a full-adder cell. The full-adder adds two input bits plus an input carry and produces a single bit sum plus an output carry (Figure 2.11). It will later be used in a bit-slice manner to build adders with word inputs, e.g. 8-bit wide.

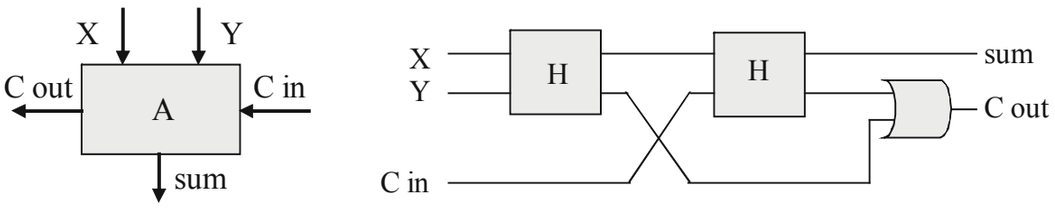


Figure 2.11: Full-Adder symbol (3-bit) and implementation

2.2 Function Units

Function units are essentially higher-level combinatorial logic circuits. This means each one of them could be represented by a set of AND, OR, and NOT gates, but using the higher level building blocks from the previous Section will help to understand their functionality.

The adder for two n-bit numbers is the first function unit we introduce here (Figure 2.12). Note that we draw fat lines to indicate that an input or output consists of multiple lines (in same cases showing the numeric number next to the fat line).

Internally, an adder is built by using *n* full-adder components, each taking one input bit each from X and Y. Note that the adder's propagation delay is *n* times the propagation delay of a bit-slice full-adder component, and so the carry bits can percolate through from right to left.

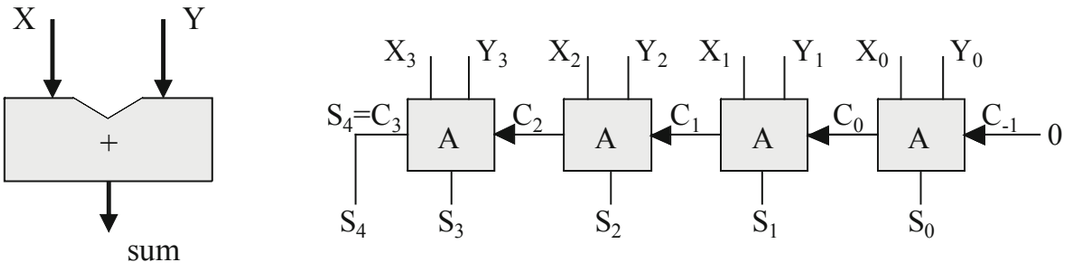


Figure 2.12: Adder function unit and implementation

Incrementing a counter by one is a standard operation for which it would be useful to have a function unit available, ready to use. Figure 2.13 shows the definition of an incrementer function unit with a single n-bit number as input and a single n-bit output. The incrementer can easily be implemented by using the adder for two n-bit numbers and hard-wiring one of the inputs to the hexa-

decimal value “$01”. By “hard-wiring” we mean to connect all “0” bits of the $01 word to electric ground, and to connect the “1” bit to the supply voltage (possibly using a pull-up resistor).

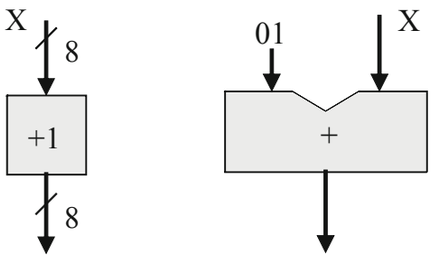


Figure 2.13: Incrementer function unit and implementation

A comparator is another very useful function unit. It takes one n-bit word as input and has only a single output line (yes or no, 1 or 0). Since in a zero-word all bits are equal to “0”, we can implement the zero-comparator by using a single NOR gate that connects to all input bits (Figure 2.14).

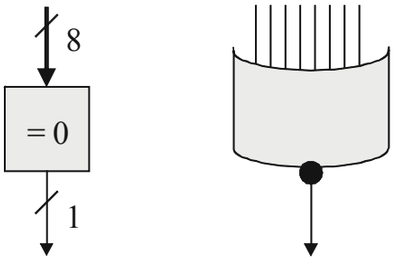


Figure 2.14: Compare with zero function unit and implementation

The one’s complement of a single input is simply the inverse of all its bits. We can implement this function unit by using *n* NOT gates (Figure 2.15).

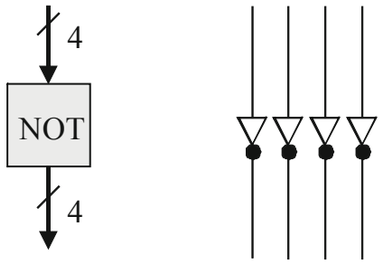
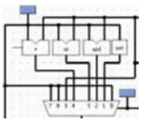


Figure 2.15: One’s complement and implementation

Having function units for AND and OR is useful and their implementation is equally simple, since each bit can be calculated independent of the other



bits. The implementation in Figure 2.16 uses *n* AND gates, each connected to the corresponding input bits from X and Y.

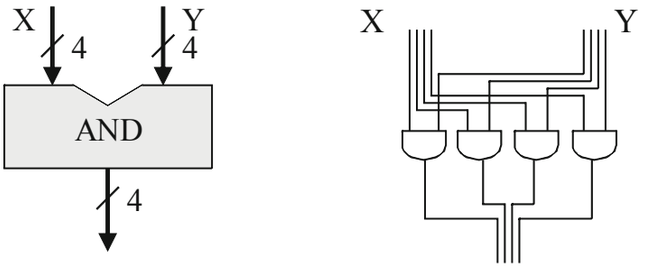


Figure 2.16: AND of two operands

The two's complement returns the negated value of an input number (Figure 2.17). We can implement this function unit by combining two of the function units we have constructed before, the one's complement (NOT) and the incrementer, executed one after the other.

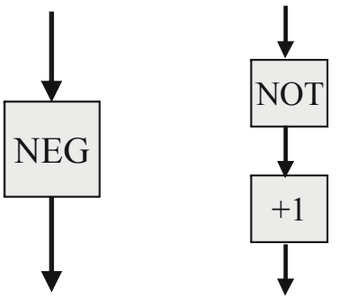


Figure 2.17: Two's complement and implementation

The subtractor shown in Figure 2.18 is another important function unit. We can implement it with the help of the previously defined function units for adding and negation.

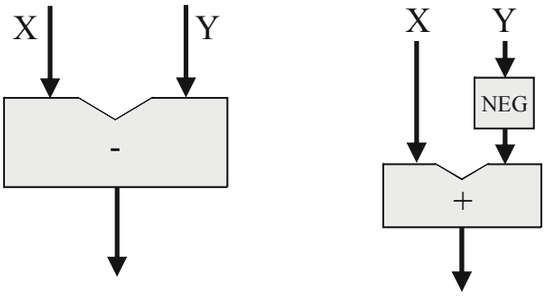


Figure 2.18: Subtractor and implementation

For a number of cases it is important to be able to compare two input numbers, e.g., to check for equality, and so we define a function unit for this, having two n-bit inputs and a single output (yes or no, see Figure 2.19). We could implement this function unit by using the previously defined function units for subtraction and check for equality to zero (Figure 2.19, middle). While this would be correct in a mathematical sense, it would be a very poor choice of implementation, both in terms of hardware components required and in the required delay time (computation time). Checking two n-bit numbers for equality can be more simply achieved by using *n* EQUIV gates (negated XORs) for a bit-wise equality check and one AND gate (Figure 2.19, right).

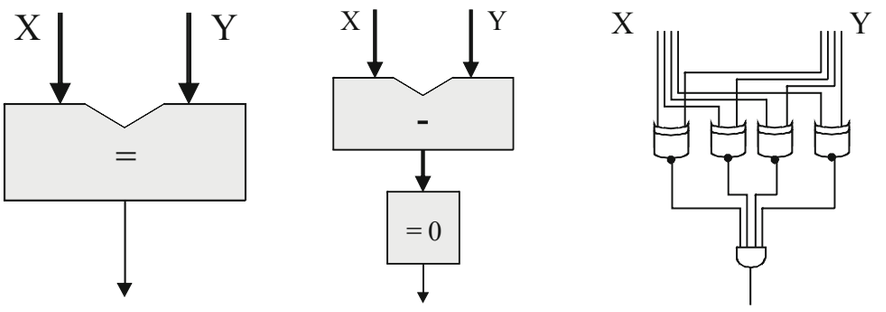


Figure 2.19: Equality of two operands and implementations

A function unit for multiplying the input number by two is another example where we have to be careful with reusing function units that are too complex for the task (Figure 2.20). Although, we could implement "multiply by two" with a single adder, the operation is equivalent with a "shift left" operation, and this we can realize with a simple reordering of the wires. No active components are required for this solution (Figure 2.20, right).

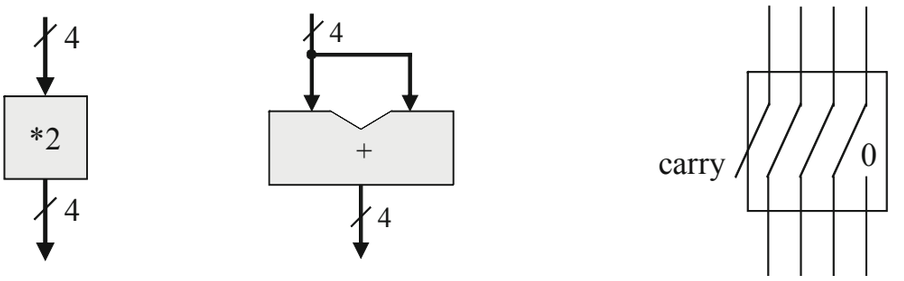
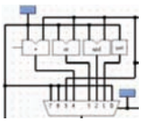


Figure 2.20: Multiply by two and implementations

Performing comparisons with integer values can be quite tricky, especially when there is a mix of unsigned and signed numbers in a system. Figure 2.21 shows a comparator that checks whether a single signed input number is less than zero (remember that an unsigned number can *never* be less than zero). In two's complement representation, the highest bit of a signed number determines whether the number is negative or positive. The implementation in Fig-



ure 2.21 takes advantage of this fact and therefore does not require any active components either.

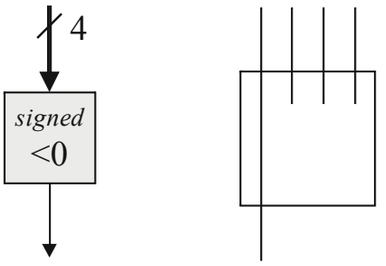


Figure 2.21: Signed comparison and implementation

We had already discussed comparing two numbers for equality, for which we had shown a simple solution using combinatorial gates. However, when comparing whether one input number is less than the other, we cannot get away with this simple implementation. For this, we do have to conduct a subtraction and then subsequently check whether the result (as a signed number) is less than zero (Figure 2.22, right).

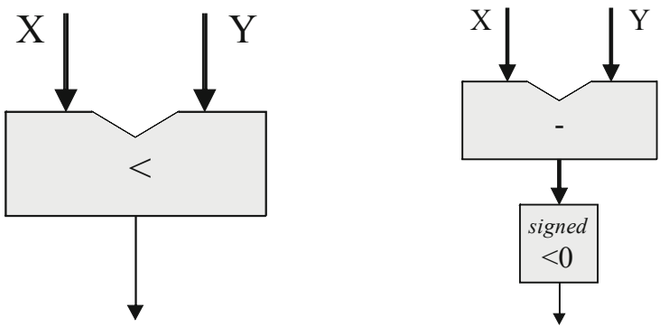


Figure 2.22: Comparison of two operands and implementation

The list of function units shown in this section is not meant to be complete. More function units can be designed and implemented using the methods shown here, whenever a specific function is considered useful for a design. The good thing about this additional level of abstraction is that we can now forget about the (hopefully efficient) implementation of each function unit and can concentrate on how to use function units in order to build more complex structures.

2.3 Registers and Memory

So far, we have been using combinatorial logic exclusively, and so a combination of AND, OR, and NOT gates, without any clock or system state. This will change when we want to store data in a register or in memory.

The smallest unit of information is one bit (short for *binary digit*), which is the information that can be held by a single flip–flop. The RS (reset/set) flip-flop type in Figure 2.23 has inputs for setting and resetting the flip-flop (both active-low in this case). The flip-flop's one-bit contents will always be displayed at output Q, while Q' displays the negated output.

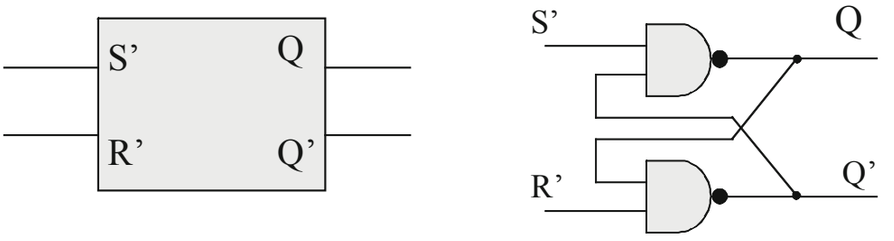


Figure 2.23: RS flip-flop and implementation

The RS flip-flop has now introduced the concept of a "state" to our circuits. Depending on whether S' or R' was activated last, our flip-flop will have the stored state "1" or "0" and will keep it indefinitely until either the set or reset input will be activated again.

One drawback of the RS-type flip-flop is that the data inputs (set or reset) are two separate lines that are "level triggered", i.e., *rising edge* (also called positive edge or low-to-high) or *falling edge* (also called negative edge, high-to-low). This means any change on these lines will cause an instantaneous change of the flip-flop contents and its output Q. However, we would like to be able to decouple the input data (as a single data line) from an "edge-triggered" activation line. These improvements can be achieved by linking two RS flip-flops together, forming a D flip-flop (Figure 2.24).

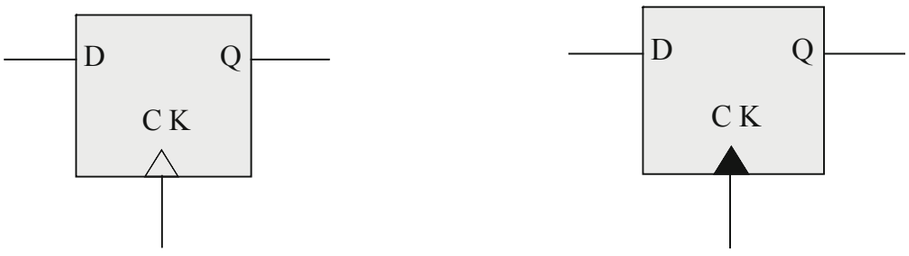
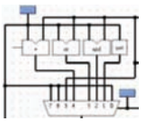


Figure 2.24: D flip-flop, positive and negative edge-triggered



The “D type” flip-flop shown in Figure 2.24 has a single data input line D and one output line Q. On the rising edge of the clock input CK, the current input of D is copied into the flip-flop and will from then on be available on its output Q. There is also an equivalent version of the D flip-flop that switches on the falling edge of the clock signal; we draw this version with a solid clock arrow instead of a hollow one.

For the D flip-flop implementation (positive edge, Figure 2.25), we use the master–slave combination of two RS flip-flops in series (the output of FF-1 is input to FF-2 via some auxiliary NAND gates), whose reset signals are triggered by opposite clock levels (inverter to the second flip-flop’s R’ input). This interlocking design accomplishes the transition from level-triggered latches to edge-triggered flip-flops. However, for understanding the following components it is more important to remember the behavior of a D flip-flop than its actual implementation.

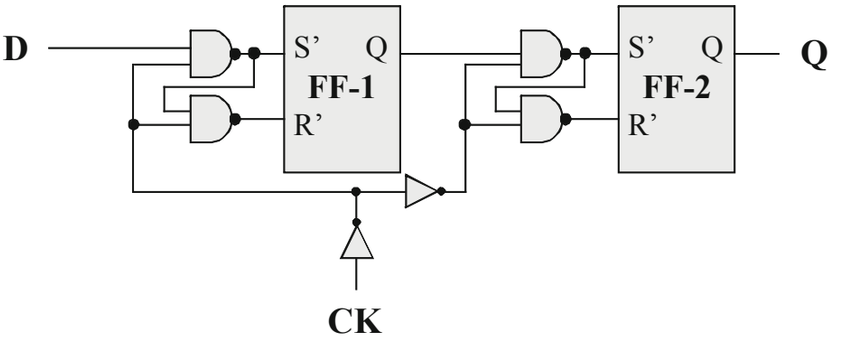


Figure 2.25: D flip-flop implementation (positive edge)

A register is now simply a bank of D flip-flops with all their clock lines linked together (Figure 2.26). That way, we can store a full data word with a single control line (clock) signal. We use a box with digits in the register symbol to denote its current contents (sort of a window to its memory contents).

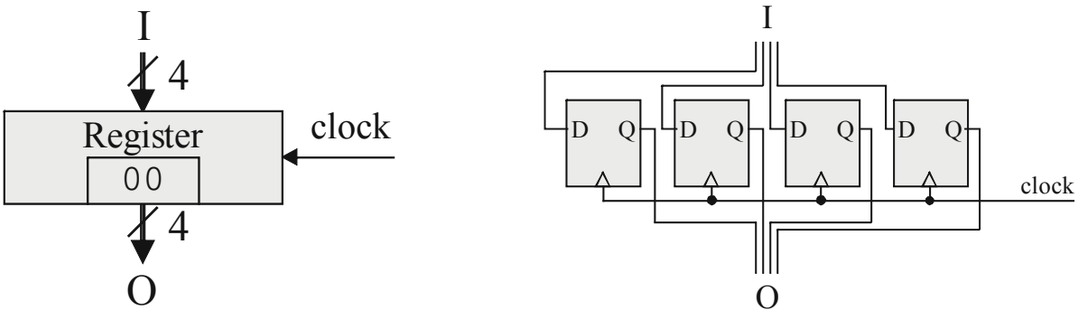


Figure 2.26: Register (4-bit) and implementation

The final components are memory modules RAM (random access memory – read and write) and ROM (read only memory) as shown in Figure 2.27. Memory modules come in various sizes, and so they will have different numbers of address lines (determining the number of memory cells) and various numbers of data lines (determining the size of each memory cell). A typical memory chip might have 20 address lines, which let it access 2^{20} different memory cells. If this memory module has eight data lines (8 bits = 1 Byte), then the whole module has 1,048,576 Bytes, which equals 1 Megabyte (1 MB).

Both ROM and RAM modules in our notation have chip select (CS', active low) and output enable (OE', active low) lines, which are required if our design has multiple memory modules or if other devices need to write to the data bus. Only the RAM module as an additional Read/Write' line (read when high, write when low) that allows data to be written back to the RAM module.

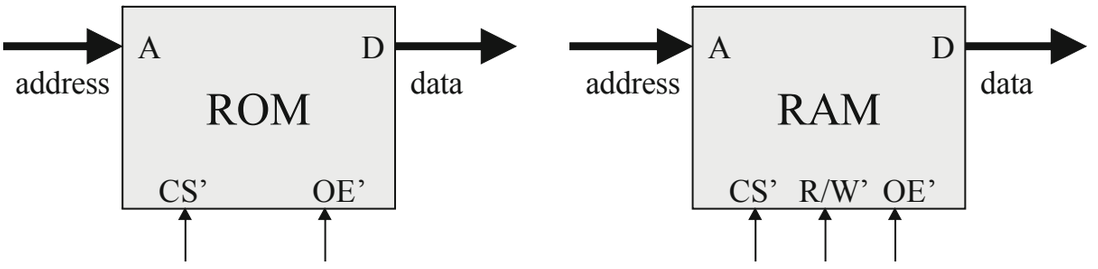


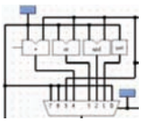
Figure 2.27: Memory modules ROM and RAM

Note that because of the complexity of memory modules, their typical delay times are significantly larger than those of simple gates or function units, which again limits the maximum CPU clock speed. At this level of abstraction we do not distinguish between different types of ROM (e.g. mask-ROM vs. flash-ROM, etc.) and RAM (e.g. SRAM vs. DRAM, etc.). It simply does not matter for our purposes here.

2.4 Retro

Before we proceed with the major CPU blocks, we introduce the Retro hardware design and simulation system [Chansavat Bräunl 1999], [Bräunl 2000]. Retro is a tool for visual circuit design at register-transfer level, which gives students a much better understanding of how to construct a complex digital system and how a computer system works in detail.

Retro supplies a number of basic components and function units (as discussed in the preceding sections) that can be selected from a palette and placed on a canvas where they will be interconnected. Components can be linked by either a single signal line or a bus of variable size (e.g. 8, 16, 32 lines). All palette components are grouped into libraries that can be loaded into the system, making Retro extendable with new component types. Retro can run in several



demo modes, displaying signal levels as colors and data in hex displays. Similar to a debugger, the simulator can be run in single-step mode and its execution can be halted at any time. Retro is implemented in Java and can run either as an applet or as a stand-alone application.

Figure 2.28 shows a sample Retro setup with the component library palette on the left and execution control buttons (VCR-style control buttons) on the top.

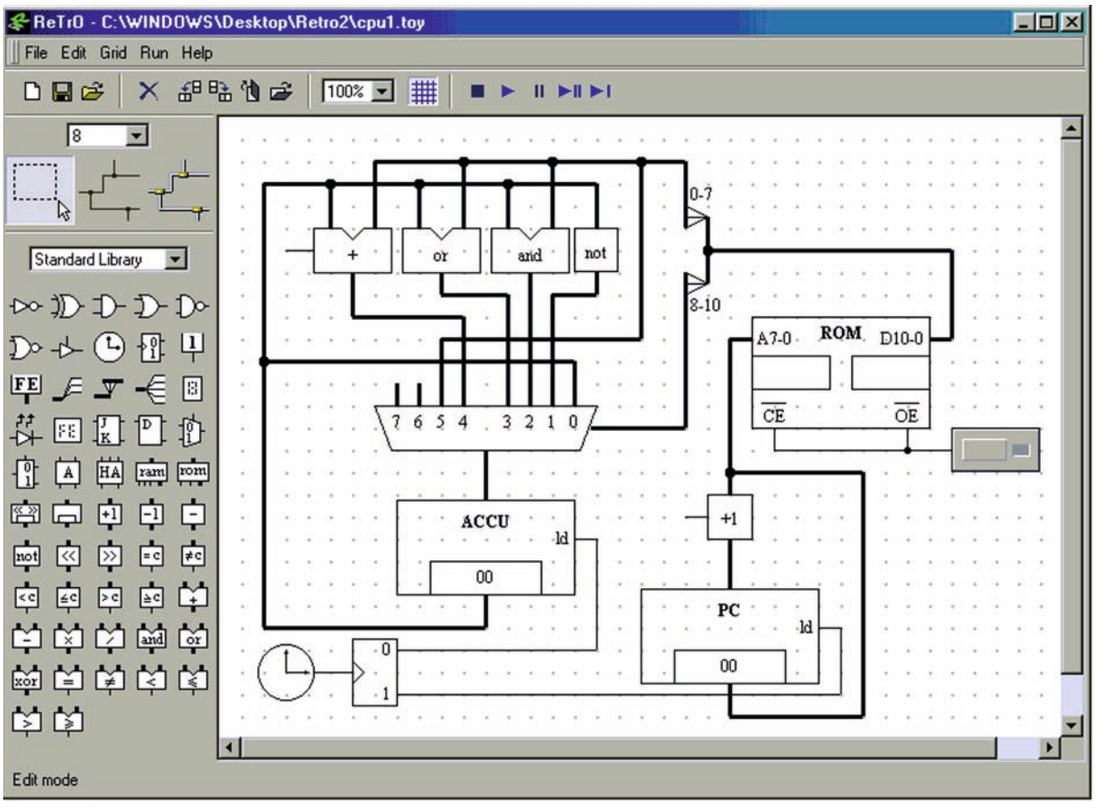


Figure 2.28: Retro simulator with library component palette

All synchronous circuits require a central clock, which is a component from the palette. The clock speed can be set in relation to the components' latencies and to the simulated time passing. Since most synchronous circuits require a number of timing signals derived from the central clock, the standard palette also includes a high-level pulse generator (Figure 2.29, left). The pulse generator has a variable number of outputs, for each of which a repetitive timing pattern can be specified.

The palette component for memory modules such as ROM and RAM are more complex than other components. They allow detailed propagation delay settings for various memory aspects and also include a tool for displaying and changing memory contents in a window or for saving to a file. Since memory data is stored in a separate data file and not together with the circuit design

data, the same hardware can be used with several different programs for individual experiments (Figure 2.29, right).

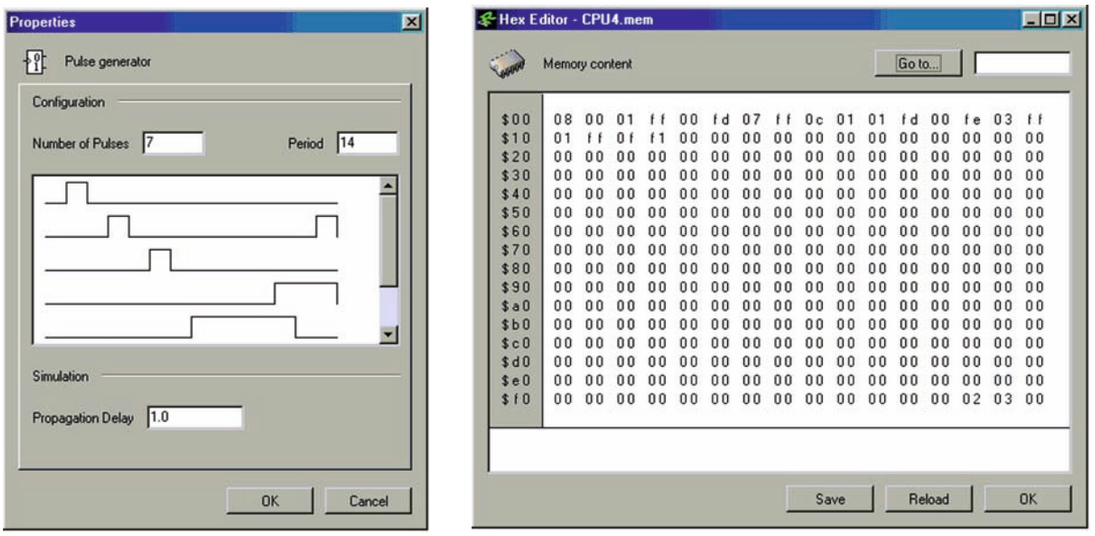


Figure 2.29: Pulse generator component and memory contents tool

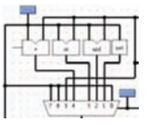
Retro was implemented by B. Chansavat under the direction of T. Bräunl [Chansavat, Bräunl 1999], [Bräunl 2000] and was inspired by N. Wirth's textbook [Wirth 1995].

2.5 Arithmetic Logic Unit

The first major component of any CPU is the ALU (arithmetic logic unit). It is the number cruncher of a CPU, supplying basic arithmetic operations such as addition and subtraction (in more advanced ALUs also multiplication and division) and logic operations such as AND, OR, and NOT for data words of a specific width. In fact, one can imagine the ALU as a small calculator inside the CPU.

One of the most important decisions to make when designing an ALU is how many registers to use and how many operands to receive from memory per instruction. For our first ALU we will use the simplest possible case: one register and one operand per instruction. This is called a *one-address machine* (assuming the operand is in fact an address – more about this later). Since here each instruction has only one operand, we need to use some intermediate steps when, e.g., adding two numbers. In the first step we load the first operand into the register (which we will call *accumulator* from now on). In the second step, we add the second operand to the accumulator.

ALUs that can perform this operation in a single step are called *two-address machines*. Each of their instructions can supply two operands (e.g. a + b) and the result will be stored in the accumulator. *Three-address machines* provide



an address for the result as well (e.g. c := a + b), and so there is no need for a central accumulator in such a system. And, just for completeness, there are also *zero-address machines,* where all operands and results are pushed and popped from a stack.

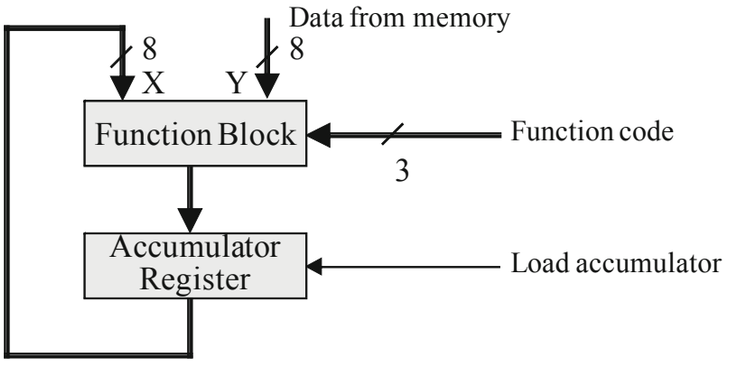


Figure 2.30: ALU structure

Figure 2.30 shows the basic ALU structure for a one-address machine. Only one operand (8-bit wide) at a time comes in from memory, and so each operation (3-bit wide) is between the accumulator (i.e., the result of the previous operation) and the operand. Also, we have made no provisions for writing a data value back to memory.

We already know what a register is, and so the remaining secret of ALU-1 is the central function block. Figure 2.31 reveals this black box. We are using one large multiplexer that is being switched by the function code (also called opcode or machine code). The 3-bit function code gives us a total of $2^3 = 8$ different instructions and each of them is defined by the respective multiplexer input.

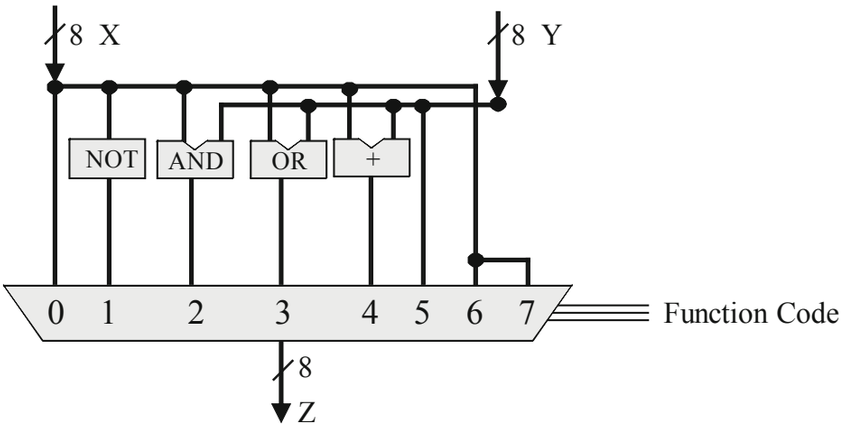


Figure 2.31: ALU function block

- **Opcode 0** simply routes the left operand through. Remember, this is linked to the accumulator, and so effectively this instruction will not change the accumulator contents. This is known as a NOP (short for *no operation*).
- **Opcode 1** negates the left operand, and so it negates the accumulator. No memory data is used for this instruction.
- **Opcodes 2, 3, and 4** perform logic AND, OR, and arithmetic addition, respectively, between left and right operand (i.e. accumulator and memory operand).
- **Opcode 5** routes the right operand through, and the accumulator will be loaded with the memory operand.
- **Opcodes 6 and 7** are identical to opcode 0, and so from the ALU point of view they are also NOPs.

It might seem like waste of resources to calculate *all* possible results (i.e. NOT, AND, OR, ADD) for every single instruction, and then discard all but one. However, since we need all of these operations at some stage in a program, there is no possible savings in terms of chip space or execution time. There may be a possible energy consumption issue, but we do not look at it now.

We can now summarize the function of these eight opcodes in a table form as machine code with mnemonic abbreviations (Table 2.2).

No.	**Opcode** (bin.)	**Operation**
0	000	Z := X
1	001	Z := NOT X
2	010	Z := X AND Y
3	011	Z := X OR Y
4	100	Z := X + Y
5	101	Z := Y
6	110	Z := X
7	111	Z := X

Table 2.2: Operations for ALU-1

2.6 Control Unit

The CU (control unit) is the second part of each CPU, enabling step-by-step program execution and branching. The central register used in the CU is the program counter. The program counter addresses the memory in order to load opcodes and operands (immediate data or memory addresses) from memory.

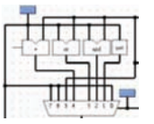


Figure 2.32 shows a first, very simple CU structure. The program counter is incremented by one in each step and its output is used for addressing the memory unit. This means, every instruction (opcode + operand) will be a single word and there are no provisions for branches. Each program on this CU will be executed line after line with no exceptions (i.e., no branching forward or backward).

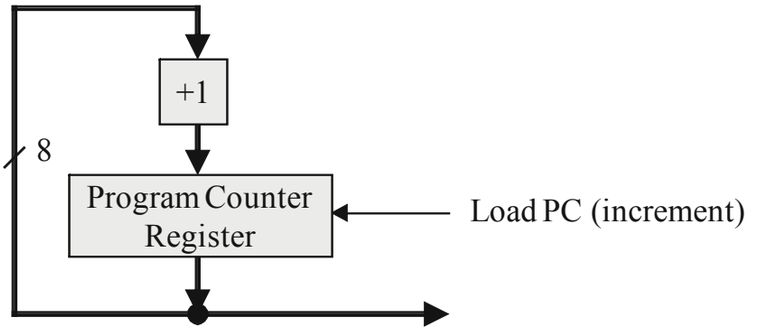


Figure 2.32: CU structure

2.7 Central Processing Unit

To build a fully functional CPU, we link together an ALU and a CU with a memory module. In this section we will introduce a number of CPU designs, starting with the most simple design, then successively adding more features when building more complex CPUs.

2.7.1 CPU-1: Minimal Design

To build the first complete CPU-1, we use ALU-1 and CU-1 from the previous two sections, linked by a ROM module (Figure 2.33).

As has been established before, this CPU design does not allow for any branching, and only immediate operands (constant values) are used in a single memory word of 11 bits that combines opcode and operand. Figure 2.34 shows the identical CPU-1 design in the Retro system (with the exception of unused or disconnected opcodes 6 and 7). The function block shows now all internal details, and the load signals for accumulator and program counter are wired up to a pulse generator, driven by the central clock.

As can be verified from the multiplexer configuration, ALU-1 supports eight opcodes, of which only the first six are being used (in the order of opcode 0··5): NOP, NOT, AND, OR, ADD, LOAD.

On the CU-1 side, the program counter (PC) always addresses the memory module and its output is fed back via an incrementer. This means, program steps are always executed consecutively, and branches or jumps are not possible.

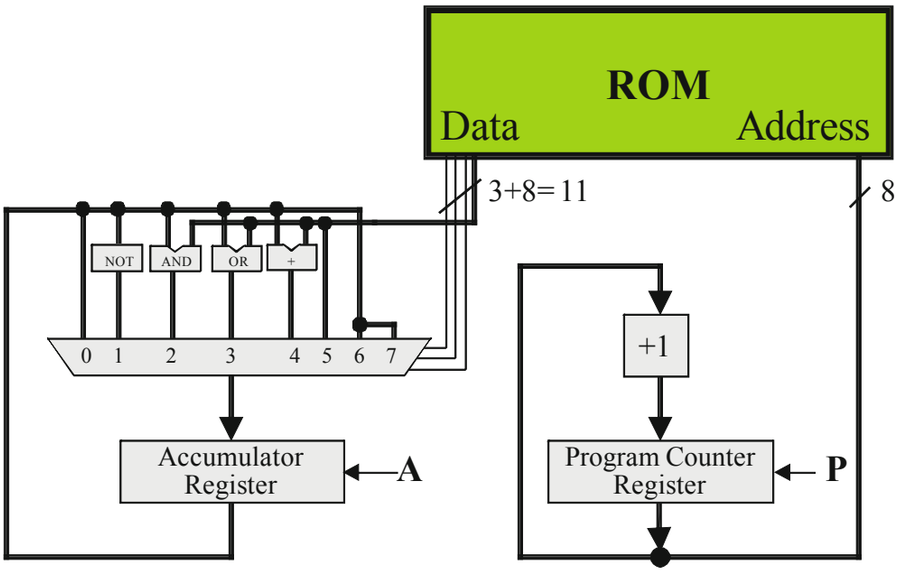


Figure 2.33: CPU-1 design

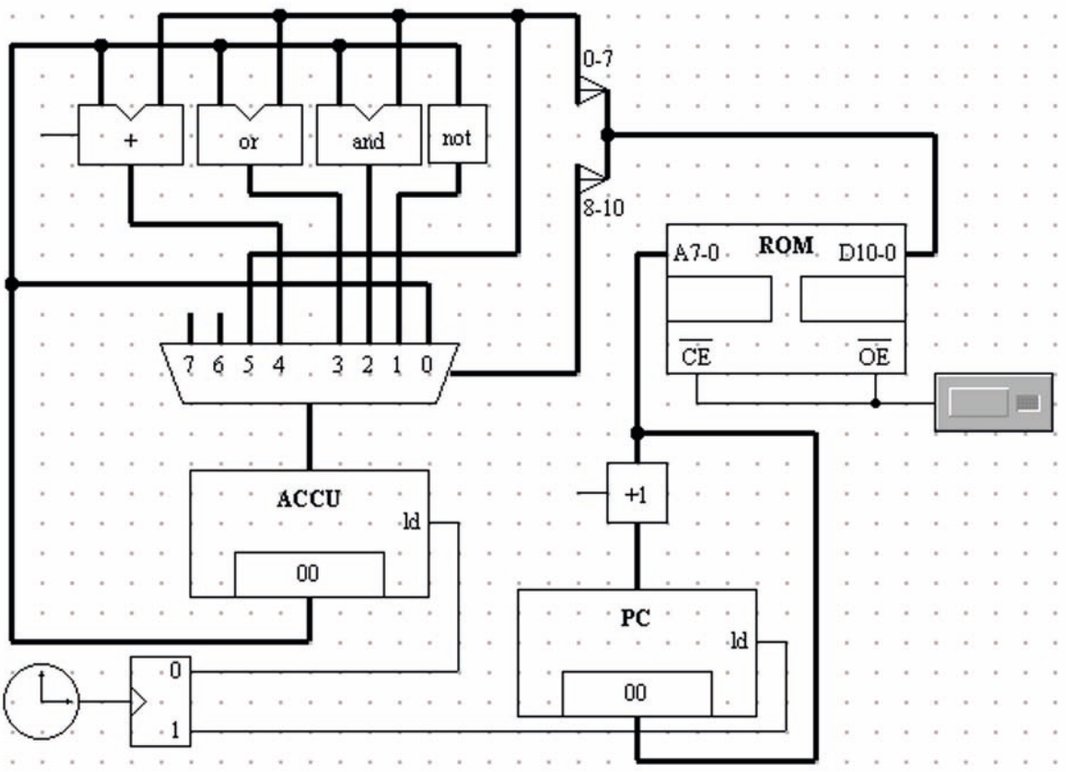
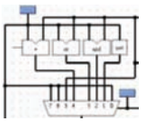


Figure 2.34: CPU-1 in Retro



The memory module uses an unusual 11-bit data format, which further simplifies the design, because operator (3-bit opcode) and immediate operand (8-bit data) can be encoded in a single instruction, and no additional registers are required to store them. The splitting of the two is simply done by dividing the data bus wires coming from the memory module, while the most significant bit of the opcode is not being used.

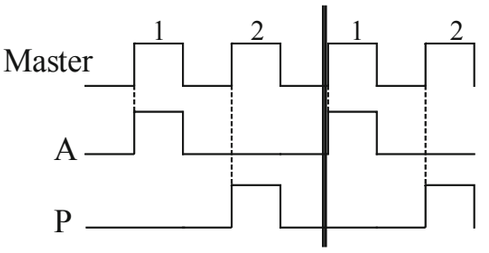


Figure 2.35: Timing diagram for CPU-1

The timing requirements for CPU-1 are minimal. Since only two registers need to be triggered, all we need are two alternating signals, derived from a master clock. First the accumulator gets triggered, then the program counter is incremented (see Figure 2.35).

Table 2.3 summarizes the available instructions for CPU-1 and lists their specific accumulator and program counter operations.

Opcode	Description	Mnemonic
0	acc ←acc pc ←pc + 1	**NOP**
1	acc ←NOT acc pc ←pc + 1	**NOT**
2	acc ←acc AND constant pc ←pc + 1	**AND** const
3	acc ←acc OR constant pc ←pc + 1	**OR** const
4	acc ←acc + constant pc ←pc + 1	**ADD** const
5	acc ←constant pc ←pc + 1	**LOAD** const
6	Not used	
7	Not used	

Table 2.3: CPU-1 opcodes

We can now look at the software side, the programming of CPU-1. We do this by writing opcodes and data directly into the ROM. The simple program shown in Table 2.4 adds the two numbers 1 and 2. With the first instruction we load constant 1 into the accumulator (code: 5 01). In the second step we add constant 2 (code: 4 02).

Address	Opcode	Operand	Comment
00	5	01	LOAD 1
01	4	02	ADD 2
02	0	00	NOP
..	..	..	..
FF	0	00	NOP

Table 2.4: CPU-1 addition program

This program also shows some of the deficiencies of CPU-1's minimal design:

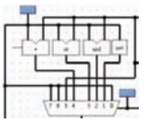
1. Operands can only be constant values (immediate operands). Memory addresses cannot be specified as operands.
2. Results cannot be stored in memory.
3. There is no way to "stop" the CPU or at least bring to a dynamic halt. This means after executing a large number of NOP instructions, the PC will eventually come back to address 00 and repeat the program, overwriting the result.

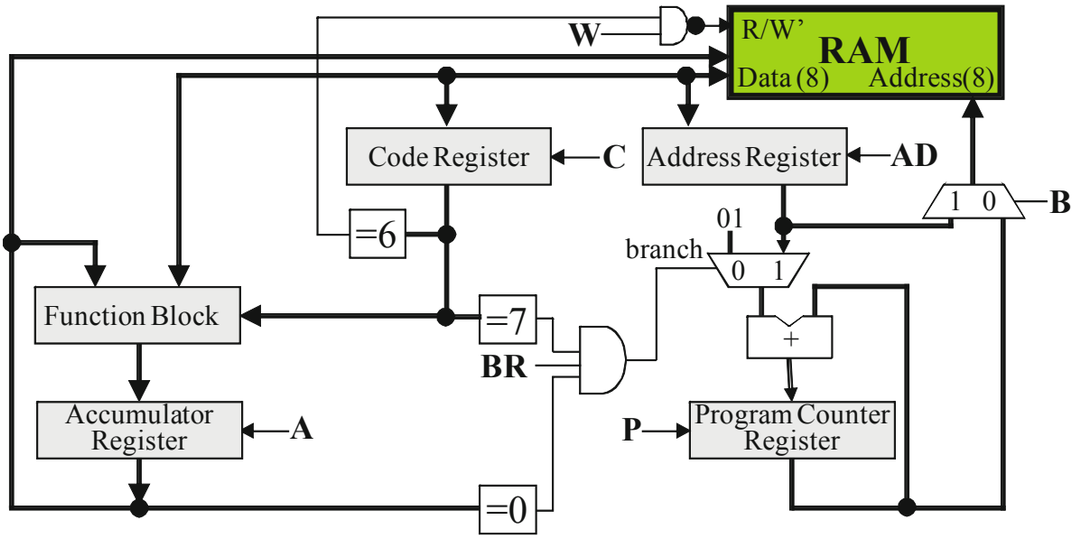
2.7.2 CPU-2: Double Byte Instructions and Branching

CPU-1 gave a first impression of CPU design, stressing the importance of timing and interaction between hardware and software. For this second design, CPU-2, we would like to address the major deficiencies of CPU-1, which are the lack of branching and the lack of memory data access (read or write).

For CPU-2, we choose an 8-bit opcode followed by an 8-bit memory address and an 8-bit wide RAM/data bus configuration. This design choice requires two subsequent memory accesses for each instruction. CPU-2 requires two additional registers, a code register and an address register for storing opcode and address, respectively, which are being loaded subsequently from memory. Figure 2.36 shows the CPU-2 schematics (top) and the Retro implementation (bottom).

The instruction execution sequence, defined by the timing diagram (microprogramming) now requires several steps:





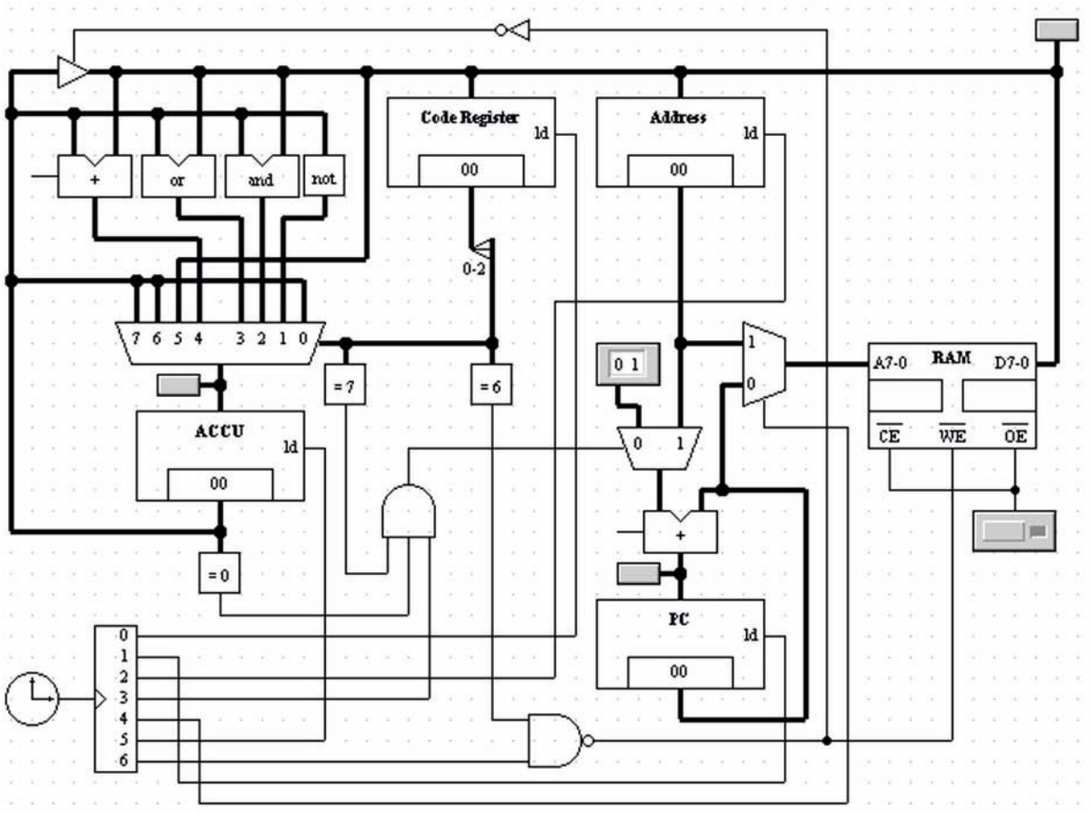


Figure 2.36: CPU-2 schematics and Retro implementation

1. Load first byte of instruction (opcode) and store it in the code register.
2. Increment program counter by 1.
3. Load second byte of instruction (address) and store it in the address register.
4. Use the address value for addressing the memory and retrieving the actual data value, which is then passed on to the ALU.
5. Update the accumulator with the calculation result.
6. (a) If required by the opcode no. 6, write accumulator data back to memory
   (b) If required by opcode no. 7, use address parameter as PC increment.
7. Increment program counter for the second time (+1 for arithmetic instructions or offset for branching).

Figure 2.37 shows the timing diagram required for CPU-2. It is important to note that the program counter will now be incremented twice for each instruction.

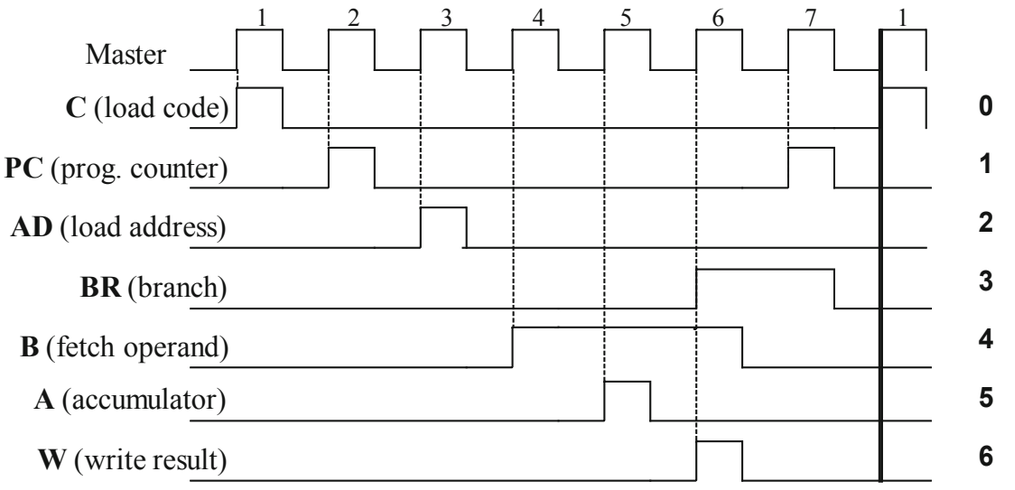
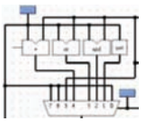


Figure 2.37: Timing diagram for CPU-2

CPU-2 uses the same ALU as CPU-1, but also makes use of the previously unused opcodes 6 and 7. As can be seen in Figure 2.36, opcode 6 is individually decoded (box "=6") and used to write the accumulator result back to the memory. Of course, this can only work if the timing diagram has made provisions for it. In Figure 2.37, we can see that one impulse of each master cycle is reserved to activate signal W, which is used to switch the RAM from read to write and also to open the tri-state gate allowing data to flow from the accumulator toward the RAM. The memory address for writing comes from the address register.

The final major addition is a conditional branch for opcode 7 (see box "=7"). The condition is true if the accumulator is equal to zero. Therefore, a branch will occur only if opcode 7 is used, the accumulator is equal to zero,



and timing signal BR is present. Signal BR overlaps (and therefore replaces) the second increment of the program counter in the timing diagram (Figure 2.37), which again demonstrates the importance of proper timing design. Table 2.5 now shows the complete set of opcodes for CPU-2.

Interaction between CPU-2's components can be seen best by following the execution of an instruction, which takes seven cycles of the master clock (Figure 2.37). Assuming the PC is initialized with zero, it will address the first byte in memory (mem[0]), which will be put on the data bus. The data bus is linked to the code register, the address register, and (over a multiplexer) to the ALU, but only one of them will take the data at a time. Since the first cycle activates signal C (line 0 from the pulse generator), this triggers "load" on the code register, and so mem[0] will be copied into it.

In the second cycle, "PC load" will be triggered (line 1). The PC's input is an adder with the left-hand side being the constant 1 (via a multiplexer) and the right-hand side being the PC's previous value (0 in the beginning). So as long as the multiplexer is not being switched over, a pulse on "PC load" will always increment it by 1. With the PC now holding value 1, the second memory byte (mem[1]) is on the data bus, and at cycle 3 (line AD), it will be copied into the address register.

Opcode	Description	Mnemonic
0	acc ←acc pc ←pc + 2	**NOP**
1	acc ←NOT acc pc ←pc + 2	**NOT**
2	acc ←acc AND memory pc ←pc + 2	**AND** mem
3	acc ←acc OR memory pc ←pc + 2	**OR** mem
4	acc ←acc + memory pc ←pc + 2	**ADD** mem
5	acc ←memory pc ←pc + 2	**LOAD** mem
6	memory ←acc pc ←pc + 2	**STORE** mem
7	(* acc unchanged *) if acc = 0 then pc ←pc + address else pc ←pc + 2	**BEQ** address

Table 2.5: CPU-2 opcodes

Cycle 4 will activate signal B (line 4), which switches the RAM addressing from the program counter to the current address register contents. This is needed since each instruction in CPU-2 has an address operand instead of an immediate (constant) operand in CPU-1 (see also opcodes in Table 2.5). With the address register now being connected to the RAM, the data bus will have the memory contents to which the address register points to. This value will be selected as input for the ALU.

With the ALU's left-hand side input being the accumulator's old value and the right-hand side input being set to the data bus (memory operand), the code register selects the desired operation over the large multiplexer on top of the accumulator. Cycle 5 then activates the accumulator's load signal (line 5), to copy the operation's result.

Cycle 6 will activate signal W (line 6). In case the instruction opcode is 6 (STORE memory, see Table 2.5), the RAM's output enable line will be activated (see box "=6" and NAND gate) and the accumulator's current value is written back to the RAM at the address specified by the address register. In case the current instruction is not a STORE, the RAM's write enable and the tri-state gate will not be activated, and so this cycle will have no effect.

Cycle 6 also activates signal BR (line 3), to not to waste another cycle. This flips CU-2's adder input from constant "+1" to the address register contents, but *only* if either the instruction's opcode is 7 (BEQ, branch if equal) and the current accumulator contents is equal to zero (see box "=0").

Finally on cycle 7, the program counter is updated a second time, either by "+1" (in case of an arithmetic instruction) or by adding the contents of the address register to it (for the BEQ instruction, if a branch is being executed).

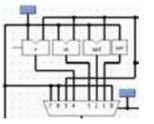
This concludes the execution of one full instruction. On the next master clock cycle, the system will start over again with cycle 1.

Example Program

Table 2.6 shows the implementation of an addition program, similar to the one for CPU-1. However, in CPU-2 there are no constant values, instead all operands are addresses. Therefore, we first load the contents of memory cell A1 (instruction: 05 A1), then add the contents of cell A2 (instruction: 04 A2), and finally store the result in cell A3 (instruction: 06 A3).

After that, we would like to bring the program to a dynamic halt. We can do this with the operation BEQ -1 (in hexadecimal: 07 FF). Although each instruction takes 2 bytes, we must decrement the program counter only by -1, not -2. This is because at the time the data value -1 (FF) is added to the program counter, it has only been incremented once so far, not twice.

We also have to consider that the branching instruction is conditional, and so for the unconditional branch we need here, we have to make sure the accumulator is equal to 0. Since we do no longer have constants (immediate values) that we can load, we need to execute a LOAD memory instruction from an address that we *know* has value zero before we can actually execute the branch instruction. In the example program, memory cell A0 has been initialized to 0 before program start.



Address	Code	Data	Comment
00	05	A1	LOAD mem [A1]
02	04	A2	ADD mem [A2]
04	06	A3	STORE mem [A3]
06	05	A0	LOAD mem [A0] *"0"*
08	07	FF	BEQ -1

Table 2.6: CPU-2 addition program

2.7.3 CPU-3: Addresses and Constants

The design of CPU-3 (Figure 2.38) is extending CPU-2 by adding:

- Load operation for constants (immediate values)
- Unconditional branch operation

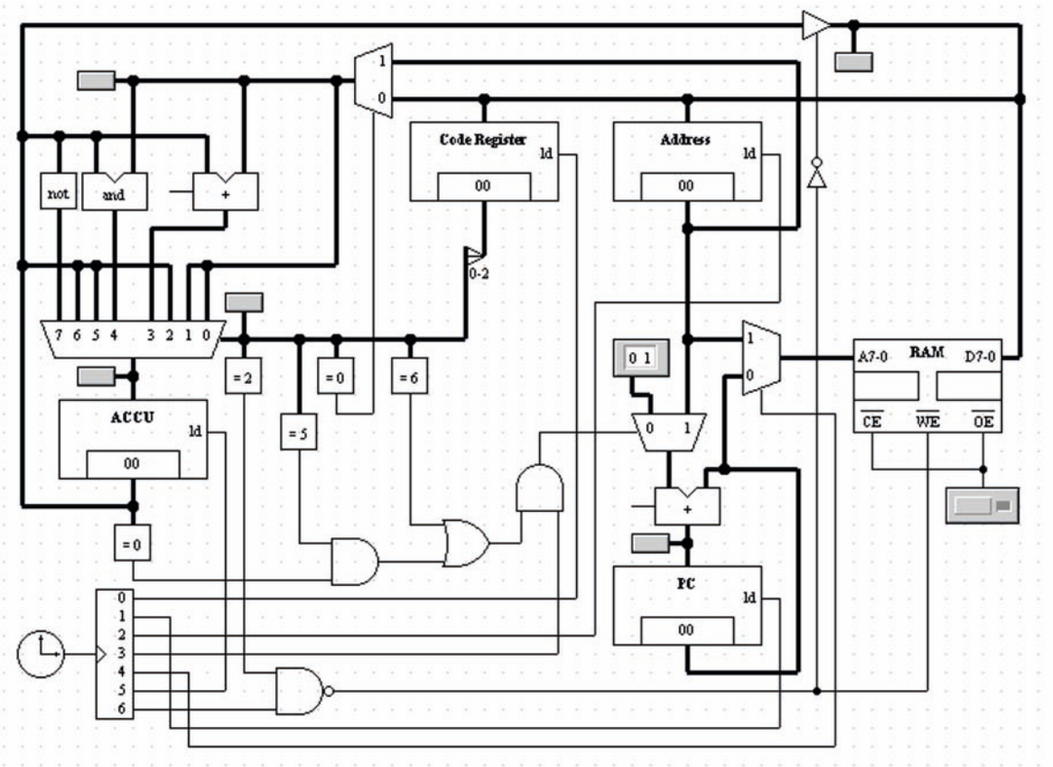


Figure 2.38: CPU-3 design

CPU-3 still uses only eight different opcodes in total (3 bits), and so we reduced the functionality of ALU-3 in this design.

An additional multiplexer, controlled by opcode 0 (see box "=0"), allows switching between feeding ALU-3 with memory output (memory operand, direct addressing) and address register contents (constant, immediate operand). However, this trick only works for the LOAD operation. Since the opcodes for ADD and AND are not equal to 0, these instructions will still receive data from memory.

The second change is to include both conditional branching (opcode 5) and unconditional branching (opcode 6). Note that the STORE instruction in CPU-3 has been changed to opcode 2. A branch will now be executed *either* (see OR gate in Figure 2.38) if opcode is 6 (see box "=6") *or* if opcode is 5 (see box "=5") and the accumulator is equal to 0.

2.7.4 CPU-4: Symmetrical Design

While CPU-3 addressed some of the deficiencies of CPU-2, its design is more an ad hoc or makeshift solution. The redesigned CPU-4 shown in Figure 2.39 shows a much clearer, symmetrical design.

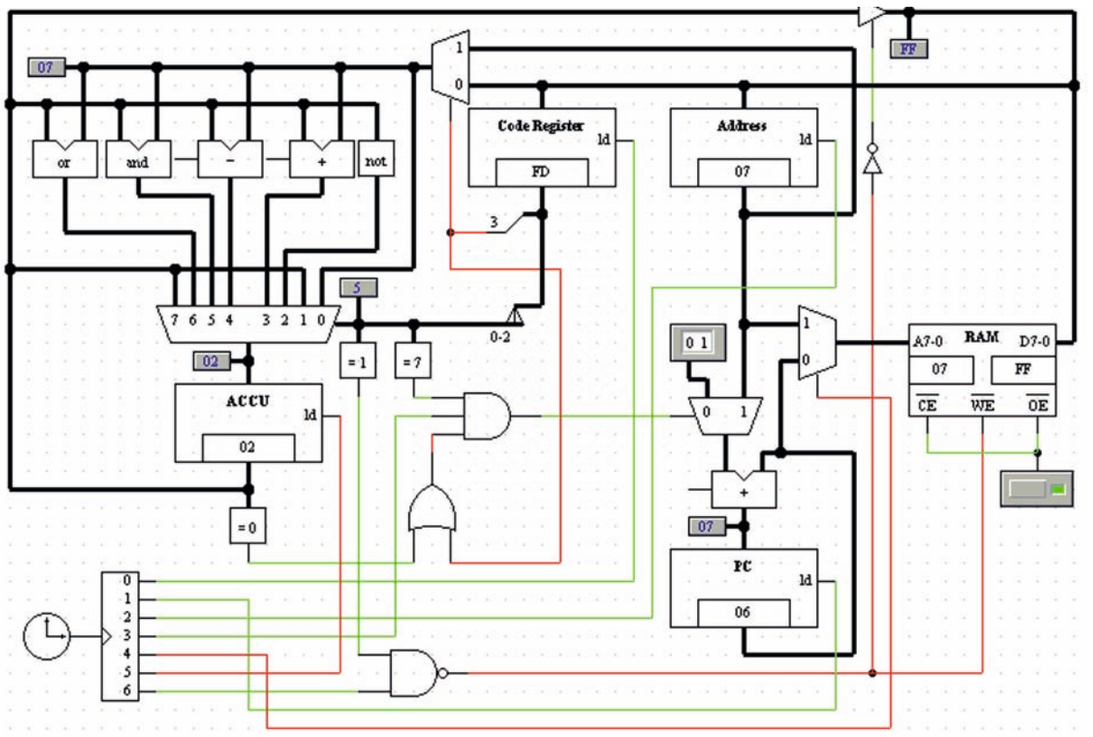
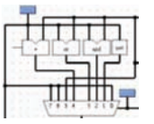


Figure 2.39: CPU-4 design



Opcode	Description	Mnemonic
0	acc ←memory pc ←pc + 2	**LOAD** mem
1	memory ←acc pc ←pc + 2	**STORE** mem
2	acc ←NOT acc pc ←pc + 2	**NOT**
3	acc ←acc + memory pc ←pc + 2	**ADD** mem
4	acc ←acc - memory pc ←pc + 2	**SUB** mem
5	acc ←acc AND memory pc ←pc + 2	**AND** mem
6	acc ←acc OR memory pc ←pc + 2	**OR** mem
7	(* acc unchanged *) if acc = 0 then pc ←pc + address else pc ←pc + 2	**BEQ** mem
8	acc ←constant pc ←pc + 2	**LOAD** const
9	Not used	
10	Not used	
11	acc ←acc + constant pc ←pc + 2	**ADD** const
12	acc ←acc - constant pc ←pc + 2	**SUB** const
13	acc ←acc AND constant pc ←pc + 2	**AND** const
14	acc ←acc OR constant pc ←pc + 2	**OR** const
15	(* acc unchanged *) pc ←pc + address	**BRA** addr

Table 2.7: CPU-4 opcodes

We are now using one additional bit for opcodes (4 bits), resulting in 2^4 = 16 instructions in total. The design is symmetrical in the way that the highest bit (bit 3) of the opcode is used to switch between constants (immediate oper-

ands) and memory operands; therefore two versions of each instruction exist. All opcodes for CPU-4 are listed in Table 2.7. The instructions belong into two groups, instructions with opcodes 0..7 use memory data (direct address), while instructions with opcodes 8..15 use constants (immediate data) instead. The distinction between memory and constant values is made by bit 3 in each opcode. The same bit line is used to distinguish between conditional branch (opcode 7) and unconditional branch (branch always, opcode 15). Bits 4..7 of each opcode byte are not used in CPU-4, but could be utilized for extensions.

As can be seen in Figure 2.39, bit no. 3 is split from the code register output and used to switch the multiplexer between immediate (constant) and direct (memory) operands for *every* opcode. This symmetric design is characteristic for good CPU designs and highly appreciated by assembly programmers and compiler code generators.

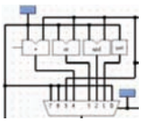
A similar solution for including conditional and unconditional branches as in CPU-3 has been implemented. Here, opcodes 7 and 15 have been selected, as they correspond to each other with respect to opcode bit no. 3 and always use an immediate address parameter, never a value from memory.

Example Program

As example program we selected the multiplication of two numbers by repeated addition (see Table 2.8). The program expects the two operands in memory cells $FD and $FE and will place the result in $FF.

Addr.	Code data	Mnemonic	Comment
00	08 00	LOAD #0	Clear result memory cell ($FF)
02	01 FF	STORE FF	
04	00 FD	LOAD FD	Load first operand ($FD) ..
06	07 FF	BEQ -1	.. done if 0 (BEQ -1 equiv. to dynamic HALT)
08	0C 01	SUB #1	Subtract 1 from first operand
0A	01 FD	STORE FD	
0C	00 FE	LOAD FE	Load second operand ($FE) and add to result
0E	03 FF	ADD FF	
10	01 FF	STORE FF	
12	0F F1	BRA -15	Branch to loop (address 4)

Table 2.8: CPU-4 program for multiplying two numbers in memory



First, the result cell is cleared. Then, at the beginning of a loop, the first operand is loaded. If it is equal to zero, the program will terminate. For halting execution, we use a branch relative to the current address –1. This will engage the CPU in an endless loop and effectively halt execution.

If the result is not yet zero, 1 is subtracted from the first operand (which is then updated in memory) and the second operand is loaded and subsequently added to the final result. The program ends with an unconditional branch statement to the beginning of the loop.

2.8 References

BRÄUNL, T. *Register-Transfer Level Simulation*, Proceedings of the Eighth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2000, San Francisco CA, Aug./Sep. 2000, pp. 392–396 (5)

CHANSAVAT, B., BRÄUNL, T. *Retro User Manual*, Internal Report UWA/CIIPS, Mobile Robot Lab, 1999, pp. (15), web: `http://robotics.ee.uwa.edu.au/retro/ftp/doc/UserManual.PDF`

WIRTH, N. *Digital Circuit Design*, Springer, Heidelberg, 1995