The Anitra Computer

A Complete Minimalist Computer System Designed, Built and Programmed at a Low Level of Abstraction

by Eirik Bakke

Anitra's two universal instructions

Computers work by mechanically executing a sequence of so-called instructions in memory, and even Anitra works this way. An instruction is a signal for the computer to carry out a single low-level computational operation. Such operations may read and alter values in memory, or affect the order instructions are executed themselves. By combining such instructions in the appropriate way, more advanced operations may be synthesized. The Anitra computer has a predefined area in memory where instructions may reside, subdivided into *blocks* of 8 instructions each. These will execute over and over again in an eternal loop. While traditional computers have hundreds of different instructions available for the programmer to use, Anitra is a "minimalist" computer and has only two:

"mov S,Q" — Read the value in memory at address S, and save an inverted copy at address Q.

"add S,Q" — Read the values in memory at address S and Q, add them together, and save the result inverted at address Q. If the result is too large to fit into Q, skip the rest of the instructions in the current block. (In the latter case, discard the result's most significant bit and save the rest. If the current instruction is the last one in its block, skip all the instructions in the following block instead.)

An address is a value identifying a single place in memory. In Anitra, an 8-bit computer, each such place may hold a value made up of 8 binary digits. A value is said to be inverted if each of its digits are complemented (1s becomes 0s and vice versa).

Although Anitra's two different instructions may seem fairly primitive, and although the area of memory where instructions are executed can only hold 128 such instructions, there is still in fact no limitation on what software can be written using them. Because the instructions can access any part of the memory, and because some instructions may be used to reprogram other ones, there are plenty of ways to extend Anitra's functionality to a more practical level. Anitra's two instructions are hence truly universal.

One of the well known classic minimalist instruction sets is that of Ross Cunniff's *One Instruction Set Computer* (OISC), with only one instruction, namely, "subtract-and-branch-if-result-negative". A comparison with Anitra's design suggests that the OISC can not only be further simplified in terms of hypothetical hardware requirements, but also that other very useful instructions such as "mov" may be added at no cost at all (simply skip fetching the second operand).

The Anitra *CPU* Hardware Registers and Signals List

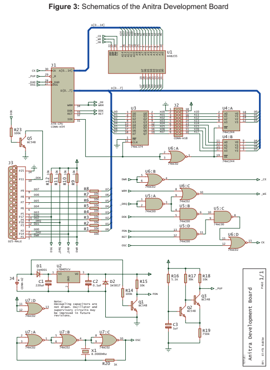
Symbol	Full Name	Description
Internal Registers		
A	Address register	Fetches *near addresses* from data bus
S	Segment register	Fetches or clears current *segment number*
PC	Program counter	Counts 8 most significant bits of the *instruction pointer*
R	Accumulator result	Stores a value for arithmetic addition
LP	End of loop detector	Used to detect whether current instruction is last in *loop*
BR	Branch if carry detector	Used to fetch any carry signal during accumulator addition
ES	Execution stepper	Determines and holds the current *execution step*
External Input Signals		
CK	System clock	Makes Anitra proceed to next execution step on positive edge
_R	System reset	Makes Anitra reset all of its internal registers when low
_DAD	Debug fetch A	Clocks A on negative edge during *debug*, should otherwise be high
DSG	Debug fetch S	Clocks S on positive edge during *debug*, should otherwise be high
External Output Signals		
A[0..14]	Memory address bus	Holds current address to be selected by memory
WRM	Write enable memory	Requests memory write when high
DOK	Debug possibility notification	*Debug* possible when high
RET	Loop return notification	Shutdown possible when high
IOX	IO exchange request	Requests *IO exchange* on positive edge/when high[†]
External Bidirectional Signals		
D[0..7]	Data bus	Conveys data between *CPU*, Mem and *external interface* (I/O)
External Other Signals		
VCC	Power	Power (+5V)
GND	Ground	Ground
Internal Signals to Registers		
_CLS	Clear S	Clears S when low
_CLR	Clear R	Clears R when low
CPA	Clear lower PC	Clears PC's 4 least significant bits when high
CPB	Clear upper PC	Clears PC's 4 most significant bits when high
_IPA	Increase lower PC	Increases PC's lower 4 bit counter on negative edge
_IPB	Increase upper PC	Increases PC's upper 4 bit counter on negative edge
_OEA	Output enable A	Tri-state activates A when low
_OEP	Output enable PC	Tri-state activates PC when low
_OER	Output enable R	Tri-state activates R when low
FEA	Fetch A	Clocks A on positive edge
FES	Fetch S	Clocks S on positive edge
FER	Fetch R	Clocks R on positive edge
PCL	Instruction pointer lower bit	Defines *instruction pointer*'s least significant bit
SLO	Segment lower bit out	Overrides current *segment number*'s least significant bit
_ESR	Reset ES	Resets ES to first *execution step* when low
_LPR	Reset LP	Resets LP when low
_BRR	Reset BR	Resets BR when low
LPK	Clock LP	Clocks LP on positive edge
BRK	Clock BR	Clocks BR on positive edge
Internal Signals from Registers		
CAR	Accumulator carry	Holds *accumulator*'s carry signal
ARQ	PC argument Q denoting bit	High when current *argument* is the last in instruction
BLK	PC block change denoting bit	Negative edge means next instruction is first in block
LOP	PC loop start denoting bit	Negative edge means next instruction is first in loop
INQ	Instruction qualifier bit	May hold current instruction's *instruction qualifier bit*
SLI	Segment lower bit in	Holds least significant bit fetched in S
E[0..3]	Execution step	Set high in turn according to the current *execution step*
_E[0..3]	Execution step complement	Set low in turn according to the current *execution step*
LPQ	LP status	May be high when current instruction is last in *loop*
_LPQ	LP status complement	May be low when current instruction is last in *loop*
BRQ	BR status	May hold carry of last *accumulator* operation
Internal Bus		
R[0..7]	Accumulator result bus	Holds current *accumulator* value

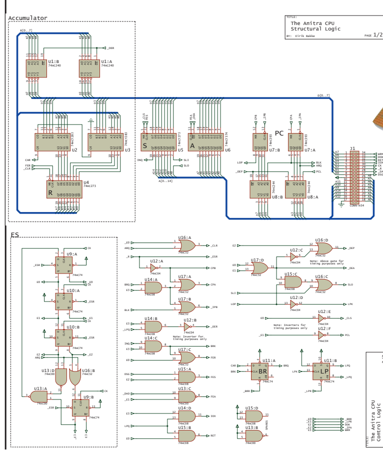
[†] The data bus will be in the input state when IOX is high. For *IO exchange*, an output value should be fetched from the data bus on the positive edge of IOX, and an input value should be held on the data bus throughout the high time of IOX.

Note: An underscore (_) in a signal name denotes an active-low or negative edge-triggered signal. X[0..n] denotes a bus X of width n bits.

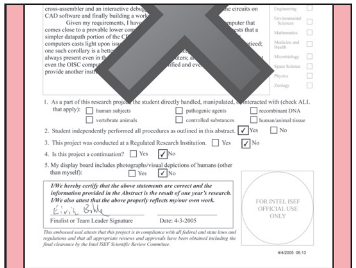
Execution step number	Tri-state activation: R or Mem selected?	Tri-state activation: A or PC (*instruction pointer*) selected?	Value of *instruction pointer*'s least significant bit	Other control signals generated
0	Mem	PC	0	Clear S Clear R
1	Mem	PC	1	Fetch A
2	Mem	A	x	Fetch S Increase PC
0	Mem	PC	0	Clear S Fetch R
1	Mem	PC	1	Fetch A
2	Mem	A	x	Fetch S Increase PC
3	None if current instruction is last in *loop*, otherwise R	A	x	Write to memory Fetch R if *instruction qualifier bit* is 1 Have PC skip to next block if accumulator overflowed and *instruction qualifier bit* is 1 Request IO exchange request if current instruction is last in *loop*
x = without significance (because PC is not selected)				

Figure 3: Schematics of the Anitra Development Board





Abstract

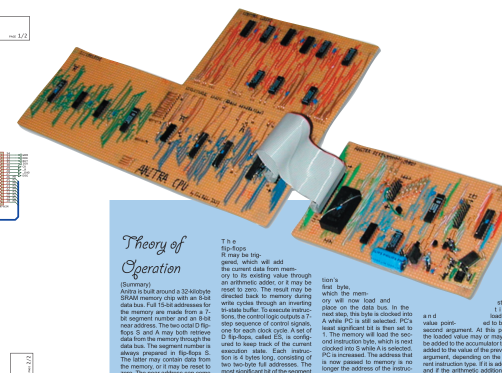


Goal and Requirements

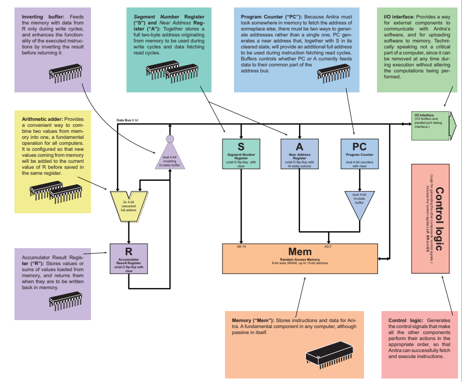
The question of this investigation is as follows: *How simple can a basic computer be designed to be, given the requirements below?*

- A computer is a machine that, given enough time and data retaining units (memory), can perform any rigidly defined computational operation on a set of input data.
- The hardware of the computer shall be designed in detail using only standard 74-series TTL-compatible digital logic circuits (ICs). A single standard SRAM chip may be used for main memory, since this requires less supporting circuitry than DRAM. Some analog support circuits may be included as appropriate.
- The design's order of simplicity is given primarily from the total number of simple logic gates logically involved in the standard circuits used, but also from the number of physical ICs and connection points involved. It should be practically possible to build a prototype of the hardware.
- The amount of accessible and addressable memory should not be the main functional limitation of the computer (see the first point). Specifically, the computer can be designed for an address space of anywhere between 2 and 64 kilobytes, inclusive.

Given the necessary level of complexity decided on after considering the requirements above, it is also a goal to get as much functionality as possible out of the available components through efficient design.



The Anitra Computer



Website: http://www.princeton.edu/~ebakke/anitra

Conclusions

Given my requirements, I have shown it possible to construct a computer that comes close to a provable lower component limit, and my investigations suggests that a simpler datapath portion of the CPU is unlikely to exist. In order to write a value to an address in memory, at least four 8-bit wide data retaining units are needed simultaneously: one to hold the value to be written to memory, one to hold the first half of the destination address while the second half is retrieved, and two for providing both parts of the address of this second half of the destination address. Interestingly, one of these registers take on the familiar role of a program counter. The computer hardware is, however, simplified by placing limits on this register's operation. The finished computer, called Anitra, is capable of executing two primitive yet universal instructions which are both based on an inverted addition operation.

The study of minimalist computers is interesting because it casts light upon issues in computer architecture design that may otherwise go unnoticed, and because it stimulates a better understanding of how software and hardware specifications interact with each other. Software and hardware engineers may not necessarily have the same perception of what is simple and not.

Theory of Operation

(Summary)
Anitra is built around a 32-kilobyte SRAM memory chip with an 8-bit data bus. Full 15-bit addresses for the memory are made from a 7-bit segment number and an 8-bit near address. The two octal D flip-flops S and A may both retrieve data from the memory through the data bus. The segment number is always prepared in flip-flops S. The latter may contain data from the memory, or it may be reset to zero. The near address can come from either flip-flops A or the counter PC, depending on which of the two currently has its tri-state outputs enabled. PC's first and last 4 bits may be increased or reset individually. PC's least significant bit is controlled directly by the control logic. A simple accumulator system is connected to the data bus. The flip-flops R may be triggered, which will add the current data from memory to its existing value through an arithmetic adder, or it may be reset to zero. The result may be directed back to memory during write cycles through an inverting tri-state buffer. To execute instructions, the control logic outputs a 7-step sequence of control signals, one for each execution step. A set of D flip-flops, called ES, is configured to keep track of the current execution state. Each instruction is 4 bytes long, consisting of two two-byte full addresses. The most significant bit of the segment number of the last address is used to distinguish between two different instruction types.

The execution sequence starts by resetting octal flip-flops R and S, and selecting PC. PC's least significant bit is set to 0. This will generate the address of the current instruction's first byte, which the memory will now load and place on the data bus. In the next step, this byte is clocked into A while PC is still selected. PC's least significant bit is then set to 1. The memory will load the second instruction byte, which is next clocked into S while A is selected. PC is increased. The address that is now passed to memory is no longer the address of the instruction itself, but the pointer that was specified as its first argument. The memory will load the value at this address, which is next clocked to the accumulator. Since flip-flops R has been reset and initially contains zero, this value will be added to the value as R is clocked this time. This sequence is then repeated to fetch the next two bytes of the instruction and load the value pointed to by its second argument. At this point, the loaded value may or may not be added to the accumulator to be added to the value of the previous argument, depending on the current instruction type. If it is added, and if the arithmetic addition operation overflows and returns a carry signal, the instruction will branch and have PC skip to the instruction in the beginning of the next instruction block. In any case, the inverse of the resulting accumulator value is finally written to memory at the same address as the last instruction argument.

Resulting Specifications from the Software Programmer's Point of View

The software programming premises that results from Anitra's method of operation are summarized below.

The Anitra computer has up to 32Kb of memory, divided into 128 256-byte *segments*. Full memory addresses consists of a *near address* and a *segment number*, denoted as from [0:0] to [127:255]. Executable instructions must be placed in the first two *segments*. Each instruction takes 4 bytes, making 128 instructions available for machine coding. The instructions will be executed sequentially in an eternal *loop*, returning to start after the last one. The first two *segments* are organized in 16 *blocks* of 8 consecutive instructions each. Branching from an instruction is done by skipping the remaining instructions in the current *block*, or when branching from the last instruction in a *block*, by skipping the complete following *block*. The instruction format is shown in Table 3.

Instruction format and argument bit patterns.

***argument* S**		***argument* Q**	
Byte 1 AAAAAAAA	Byte 2 xBBBBBBB	Byte 3 CCCCCCCC	Byte 4 iDDDDDDD
near address	*segment number*	*near address*	*segment number*
x = don't care			

The *arguments* are two full addresses. All instructions work by processing the values at memory locations [BBBBBBB:AAAAAAAA] (called S) and [DDDDDDD:CCCCCCCC] (called Q). There are two instruction types, distinguished by i, the *instruction qualifier bit*. In addition, the last instruction in the *loop* is altered to serve as a special IO instruction. The resulting instructions are given in the Table 4.

Anitra's machine code instructions.

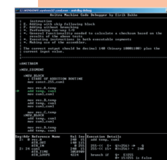
Qualifier	Instruction		Operation
Bit i=0	mov S,Q	(move with complement)	Q:=255-S
Bit i=1	add S,Q	(add with complement)	Q:=255-(S+Q **mod** 256) branch if S+Q>255
Bit i=0 + instruction is last in *loop*	iox S,Q	(exchange input/output, don't care S)	output register:=Q Q:=input register

Testing and software development

For the purpose of developing Anitra software using an ordinary desktop computer, I have written a cross-assembler, which translates assembly language code into a binary memory image, a debugger/emulator, which inputs the image and interactively simulates the software's operation on the Anitra computer, and a parallel port uploader, which transfers the image to the Development Board's memory chip. These tools make the programming process similar to that of any modern computer or micro-controller.

The project's most important piece of Anitra software is the Debug Routine. It tests all distinct aspects of Anitra's operation by running a sequence of tests that all result in different numerical answers, and then outputs the sum of all results to the user. Since the tests are designed to give a different result if Anitra does not behave according to specification, the precence of the expected sum on the output is very likely to indicate a working model. The routine was used in all development stages: first to test the operation of the emulator, then to test circuit simulations on CAD software, and finally to to test the physical prototype.

Another piece of interesting Anitra software is the Virtual Machine Emulator. The routine executes virtual instruction of another, hypothetical, computer. The virtual machine is far more advanced than Anitra itself, with 14 instructions, in-built function calls, separate data and return stacks, relative local variable addressing, unconstrained branches and so on. Although at a cost of speed, this allows Anitra to be programmed without any of the initial limitation on code size, branching etc. Another interesting observation is that the two simple instructions provided by Anitra seem to be perfectly suffient for solving common programming tasks. The code is fairly compact, and there is plenty of space for emulating more virtual instructions, or possibly, to emulate a 16-bit machine instead.



A Lower Limit

I have tried to find a find, by logic, a set of components that cannot be omitted from the computer architecture. This is feasible due to the specificness of this investigation. While the argument below is not meant to be of a mathematically satisfactory standard, it did provide is a very good starting point for the actual design. The premises of the argument are the basic requirements given in "Goal and Requirements".

1. The computer must have access to memory
Following the requirement of including between 2 and 64 kilobytes of main memory, we must include an SRAM chip; the data-retaining components found in the 74-series cannot hold more than single bytes at a time. We will have to choose among standard data and address bus widths. Of practical reasons, the data bus width is already more or less given to be 8 bits, since 4- and 16-bit standard memories fall outside of our desired size range. This leaves us with an address bus that should be between 11 and 16 bits wide, inclusive. A full address will in any case not fit in a single data bus width. For the moment, we can decide to use an address bus of 16 bits, which is exactly twice the size of the data bus. This will simplify later parts of the discussion.

2. The computer must be able to combine data
A working computer must in some way or another be able to logically combine two values from memory into a single value that can be written back into memory. Without such a capability, there would be no remotely practical way of implementing arithmetic functions in software. All bits on the bus must be included in this operation; if some of the bits are not, they will be logically inaccessible for all arithmetic functions and hence wasted memory. Hence, we must include in our design the appropriate logic for combining two values of data bus width.

Combining two values can either be done with simple two-input gates or with arithmetic full adders. The number of ICs will be the same for both techniques, but an adder configuration is far more logically complex. In a minimalist computer design, it is not altogether obvious why the relatively complex addition operation should be worthy of inclusion. However, the addition operation has one essential property that justifies its use: it has the ability to logically combine not only two values of several parallel bits each, but also to combine the individual bits within these values through a carry mechanism. [illegible]

3. The computer's software must have access to memory
A computer must be able to store any value at an address in memory that originates completely from memory itself. [illegible]

4. The computer's various components must work together
To function as a computer, the various required components must interact in the correct order, and a certain amount of logic is needed to generate the appropriate control signals for these. [illegible]