On the 6502
A brilliant or sloppy design?

Jesús Arias

March 28, 2011

Contents

1 Introduction

The 6502 was a very popular CPU during the 8-bit craze 30 years ago. It is still being used as a cheap core in the embedded market, mainly in its 16-bit incarnation: the 65816, but it is not widely known as it was decades ago. It still has many enthusiasts doing projects with it. It has been implemented as a VHDL or Verilog core many times. Several TTL CPUs had been designed using the same instruction set as the 6502, etc, etc, etc. The 6502 is a very interesting CPU due to its simplicity: few registers, few instructions, few transistors. It is a logical introductory example for computer design. And it is also interesting due to the

prominent role it played during the development of the early personal computer, with many classic machines running on it.

But, on the other hand, some of these enthusiast seems to have been so focused on it that they think it is the Holy Grail of computing, while, the 6502 is also well known for its many bugs, limitations, and weird behavior. In this document I'm mainly playing the Devil's Advocate against the 6502, exposing its not so brilliant aspects. I don't want to reduce the merit of their designers. They did a magnificent job, but their big goal was to design the cheapest possible CPU in the World, not the fastest, nor the cleanest, nor the most robust. What gave the 6502 its place in the computing history was its low price. We mustn't forget that.

This document includes the knowledge I collected from my own experimentation with a 6502 prototype I build about one year ago along with that available on Internet resources. In some occasions I'm presenting diagrams inferred from known facts, but not accurate in the stricter way. There are also some personal views that can be shared or not, specially those related with the instruction set of the processor.

The document begins with some hardware related issues of the 6502, continues discussing the limitations of its instruction set, then, a performance comparison between the 6502 and its main competitor, the Z80, is presented, and, finally the 6502 and a RISC cpu design are compared.

2 Hardware flaws, trade-offs, bugs...

The 6502 as it was originally released included lots of flaws and bugs [4]. Many of these problems were corrected in the CMOS version, the 65C02. But the improved version came to the market too late and almost all 6502 systems were designed around the NMOS chip with all its problems. It is quite sure that the designers were aware of some of these problems before the first chip went out of the factory line. Some were the result of design trade-offs when reducing the hardware complexity, while others weren't just taken seriously. None of them was considered of enough importance for a design revision, so, all NMOS 6502 exhibit them.

Here are some examples of odd hardware behavior or just missing expected functionality.

2.1 Lack of 3-state address bus

The original 6502 came without the possibility of disabling its address output drivers, and, therefore, it became a nuisance for the designers of systems with any sort of DMA. This feature can be added externally, using 74LS244 3-state drivers, but any other CPU on the market gives you this possibility for free.

The fact is that including a 3-state capability to the address bus requires only a negligible amount of hardware inside the chip, and there are many unused pins in the package, so, I don't see the reason for not including it. In fact, a follower chip, the 6510, included a 3-state address bus, probably due to customer demands (and customers were Atari and Commodore, not you or me).

2.2 Dummy Reads

The 6502 also lacks an output to validate the R/W signal: some sort or VMA or MEMRQ pin. As a consequence all clock cycles are memory accesses, either reads or writes, but, there are some cycles when the CPU is doing internal processing and the value on the data bus is irrelevant. Most of these cycles are dummy reads.

Lets consider for instance the RTS instruction whose timing diagram [2] is shown in Figure 1. Before retrieving the return address from the stack the stack pointer has to be incremented. During this cycle a dummy read is performed. The same happens at the end of the instruction when the PC has to be incremented. The RTS instruction does two dummy reads (and one dummy fetch, discussed later).

Dummy reads aren't wasted time. This time is needed for internal processing anyway. Their only impact is on memory sharing systems and in the total power consumption due to extra memory activity.

2.3 Dummy Writes

Some instructions can also have dummy writes. This happens with all Read-Modify-Write instructions like INC zp. In these instructions the data from memory is read, written to the same address unmodified and,
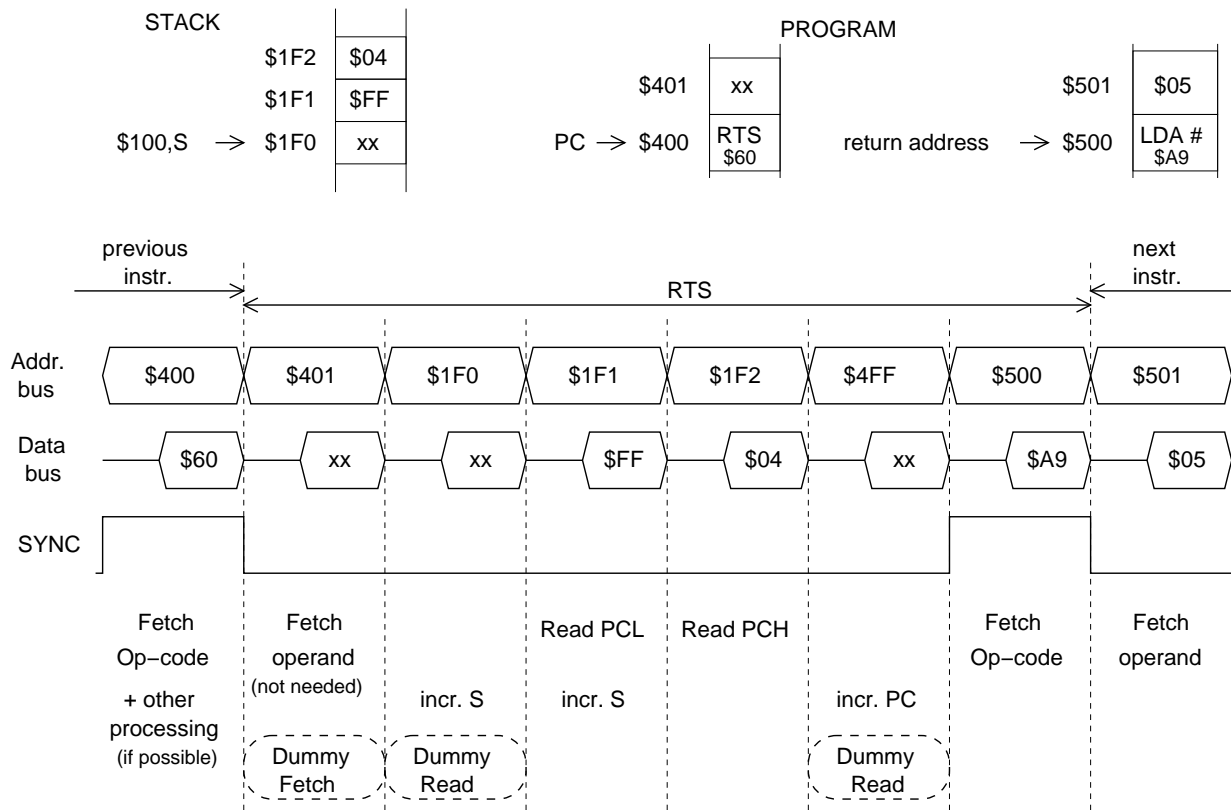
Figure 1: Detailed timing for the RTS instruction (inferred from available data).

finally, written to the same address modified [2]. The first write cycle is, thus, a dummy write. As with dummy reads, dummy writes aren't wasted time, just unnecessary memory bandwidth.

2.4 Dummy Fetches

Dummy fetches happens when a single byte instruction is executed. After reading the op-code at PC address, a second read is performed at PC+1. This second read retrieves the first byte of the operands for multi-byte instructions, but, for single-byte instructions it is unneeded, and therefore, it is wasted time.

Thanks to dummy fetches all instructions have the same first cycle. This can, surely, help in reducing the complexity of the CPU sequencer at the expense of some processing speed penalty. The 6502 was designed to be cheap and hardware simplicity was more valuable than speed. This speed penalty is also discussed in section 4.1.4 where it was found to be of little importance (about 10% slower).

Dummy fetches can explain why after a SYNC cycle the address on the bus is always incremented, with hardware interrupts being the only exceptions to this rule.

2.5 Incomplete decoding logic.

The 6502 sequencer diagram is shown in Figure 2. This diagram was obtained from data available from [3] and the transistor-level schematic from [1]. It is basically a seven-bit shift register with a "walking one" driving the AND-plane of a half PLA, the OR-plane of the PLA being replaced with random logic. PLA is not a very convenient name because the array connections aren't programmable: a connection is made by placing a transistor in the array during the design of the chip. The 6502 sequencer is therefore completely hardwired. The shift register is used for counting the instruction cycles in place of a more conventional counter and decoder. The shift register is not the only input to the PLA, other bits come from the instruction register (the register where the op-code of the current instruction is stored). The PLA AND-plane outputs are the Boolean products of any desired combination of inputs, but not all combinations can fit in the PLA. In this respect the PLA differs from an ordinary ROM. The PLA was therefore programmed to decode only the documented op-codes of the 6502 and no more than that. As undocumented op-codes were supposed to be "don't care" cases in the Boolean equations of the PLA, the size of the PLA was reduced to a minimum.
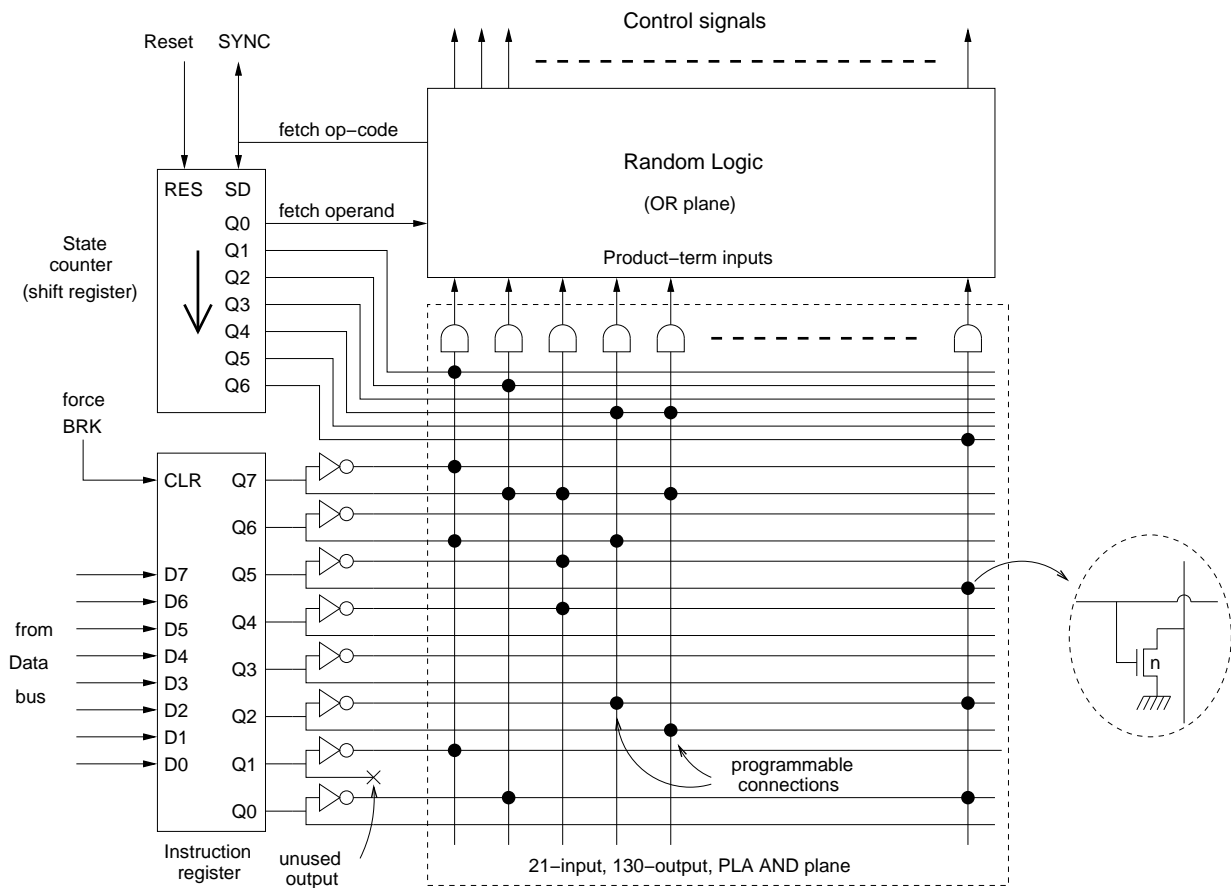
Figure 2: Inferred block diagram of the 6502 sequencer. (The programmable connection locations does not correspond to the actual ones).

Unfortunately, this design strategy also led to weird behavior when undocumented op-codes are executed. Most of them execute useless operations. Some of them have variable effects because of bus contentions inside the CPU, and a few of them have the surprisingly effect of stopping the CPU completely. These later op-codes are, jokingly called, the KIL instructions.

A KIL op-code basically don't get the SYNC output active for any of the 7 clock cycles it takes for the "walking one" to get out of the shift register. After this happens the shift register is completely filled with zeroes and the processor is dead. Only a RESET can get the 6502 into an operational state again.

This flaw could be related to pipelining, a novel concept applied to microprocessors at the time of the 6502 design. The idea is to do the op-code fetch of the new instruction in parallel with the last execution cycle of the current instruction, if possible. A convenient way of arranging this is to actually do the op-code fetch during the last cycle of the current instruction along with its last processing step. So, the SYNC signal (op-code fetch) is activated from one of the outputs of the sequencer instead of the state counter. If the current op-code fails to activate the SYNC signal it becomes a KIL.

All KIL op-codes have the bit 1 set (for instance op-code = $02). This is surely related to the fact that one output from the instruction register is not connected to the PLA (bit 1, of course).

The KIL op-codes seems to be closely related to the Halt and Catch Fire, HCF, instruction of the 6800 CPU. But, as long as I know, these op-codes were intended as a factory test, so, they were intentional. The HCF stops executing instructions and keeps the address bus counting, turning the 6800 CPU into no more than an expensive binary counter.

2.6   Interrupts & Reset

2.6.1   The B flag

One of the most bizarre things about the 6502 is the behavior of its Break flag. A "PHP, PLA" sequence always reads it as "1", but it is pushed as "0" into the stack when a hardware interrupts happens. The
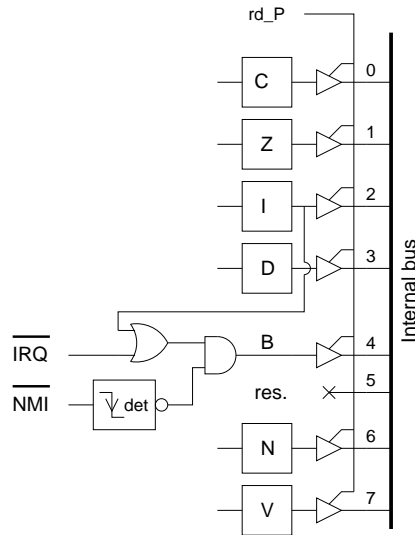
4

Figure 3: Functional schematic of the status register of the 6502 CPU

schematic of Figure 3 helps to understand this behavior: First, there is no storage for this flag. It is just the validated interrupt line. During normal program execution it is always read as "1" because a "0" will interrupt the program before the actual read. When an interrupt is executed the flag register is read with the B flag as "0" and pushed into the stack. Before jumping to the interrupt vector the I flag is set and the NMI edge detector is reset, so, when the execution continues the B flag is one again.

2.6.2   The BRK op-code ($00)

The BRK op-code can be fetched into the instruction register because of four different possible events:

1. The program contains a BRK instruction and it is fetched like any other op-code.

2. The IRQ input goes low and the I flag is reset.

3. A falling edge in the NMI input happens.

4. The CPU is reset.

The instruction register can be cleared instead of fetching the current op-code when an interrupt or reset happens, effectively converting the fetched op-code into a BRK (see Figure 2). But, then, the execution of the BRK instruction differs depending on the cause of the BRK in the following ways:

- A software BRK lets the PC to be incremented two times before pushing it into the stack, pushes the flags register (with B as "1"), and, finally, reads the new PC value from the addresses $FFFE and $FFFF.

- An IRQ interrupt pushes the PC and the flags register with B as "0", but it does not increment the PC, allowing the interrupted op-code to be fetched again after RTI. The new PC value is read from the addresses $FFFE and $FFFF.

- An NMI interrupt does the same as the IRQ interrupt but the PC is read from addresses $FFFA and $FFFB.

- The Reset is very interesting. The stack pointer is decremented by 3, like if three values were being pushed into the stack, but nothing gets written into the memory. In fact, the BRK instruction tries to push the PC and the flags register, but the R/W line is forced high and the three write cycles are turned into dummy reads. It, finally, reads the new PC value from addresses $FFFC and $FFFD.

So, the BRK instruction gets a lot of different uses, with the software BRK being the least priority to designers. Its behavior is modified with a few gates that can inhibit the normal PC increment or memory

write. The internal buses are precharged high, and if nothing pulls their lines low they will be read as all ones. To generate the three different addresses for the vectors only three pull-down transistors are needed (for bits 0, 1, and 2). A lot of functionality is achieved with only a few transistors. Compare this to the burden of the 8080 case where the external interrupt source has to put an op-code into the data bus.

2.6.3 Lost BRKs

The BRK instruction is like a fixed address subroutine call. The only noticeable remark being the fact that it is actually a two-byte instruction. The second byte is not used by the processor, but it can be retrieved from the program memory by the BRK handler routine and it can get a user-defined meaning. It is tempting to use the BRK as a system call, but beware: the 6502 has an important bug regarding the BRK instruction [4]. Interrupts are executed by turning the currently fetched op-code into a BRK. But, when the interrupted instruction is also a BRK the PC increments like for a software BRK but the B flag is pushed as "0" like for a hardware interrupt. As a consequence, the BRK handler is executed for a hardware interrupt, and, when the RTI instruction is executed, the next instruction fetched is that after the BRK. The BRK instruction is therefore skipped, like if it was removed from the normal program flow.

2.6.4 No instruction between interrupts

The IRQ handler routine has to take the necessary steps in order to deactivate the IRQ line before returning to the interrupted code with an RTI instruction. If the IRQ line is still low when the RTI is executed a new interrupt will happen just at the end of the RTI execution. Not a single instruction of the interrupted program is executed in this case. This differs from the behavior of other processors where one instruction of the interrupted program gets executed between interrupts, a trick often exploited by debuggers to implement a single-step execution. In the 6502 case other solutions have to be found, like, for example, triggering an IRQ just after the fetch cycles of the executed instruction by using a carefully set timer (if your system includes a 6522 VIA this is possibly the simplest solution).

2.7 BCD arithmetic
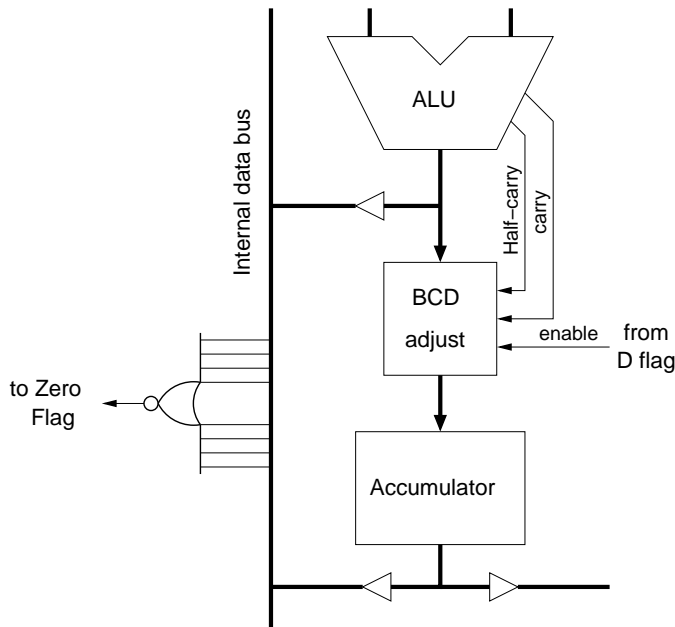
Figure 4: Detail of the BCD correction block and its placement in the 6502 datapath.

The 6502 is able do perform arithmetic using BCD values. Instead of using a BCD adjusting instruction like many other processors (namely DAA on Intel's or Z80), the mode of operation is selected by the decimal flag in the status register. If the D flag is one the ADC and SBC instructions operate in BCD mode. For the ADC instruction this involves dividing the 8-bit data into two 4-bit BCD digits and to add 6 to the nibbles whose value exceeds 9, a condition that requires one carry output for each 4-bit digit of the ALU. The BCD

adjust are two simple 4-bit adders that can add 6, 9 (for SBC) or 0 to each digit depending on their control inputs. The Figure 4 shows a diagram of the ALU, BCD adjust logic, accumulator, and the way they are interconnected.

When operating in decimal mode the N and V flags doesn't make sense, but the Z flag is also invalid. That's odd, because the Z flag can have an useful role also for BCD values, but, the real 6502 can have the Z flag set when the result is $66 in decimal mode.

This flaw is easily explained by looking at the diagram of Figure 4. The Z flag is computed as the 8-bit NOR function of all the bits of the internal data bus, not the accumulator inputs. Thus, an active Z flag is telling us that the output of the ALU is zero but the BCD adjust logic can have added some non-zero value to this result. It seems that the Z flag computation is done in the wrong place. But, that place was selected because there are instructions that can change the Z flag without having the ALU nor the accumulator involved (for instance LDX).

So, it looks that the 6502 designers didn't want to have the Z flag correct. Doing this would have required another 8-input NOR gate placed on the accumulator inputs and the Z flag source switched depending if the accumulator is the destination or not. This doesn't look like much extra hardware, but the designers went the easy way: declaring the Z flag invalid when in decimal mode. It seems that they had little regard for this mode, maybe because it was a last hour marketing decision to boost sales by offering something more. In my opinion the BCD mode has little or no practical use, and, by the way, the developers of Commodore should have had the same opinion because they forgot to clear the D flag in the IRQ handler routine of the C64 [4] :) Modern CPUs no longer have BCD support, further supporting this opinion.

3   Instruction set

The 6502 got an "spartan" instruction set that made things a little difficult for programmers. This instruction set was increased in the 65C02 with much needed instructions. As always happens with instruction set upgrades, applications usually target the smaller instruction set in order to run on all possible CPUs, and, as a consequence, improvements aren't used in most of the actual code. This is particularly true for the 6502: Almost all systems were based on the NMOS 6502 and only a negligible amount of code was optimized for the newer CMOS version.

Both the oddities of the 6502 instructions and their addressing modes are presented in the following text.

3.1   Useless addressing modes

The 6502 fans are proud of the many addressing modes it provides to programmers. But, some of them have little or no practical use. The ZP,X and ZP,Y are seldom used because arrays aren't placed on zero page very often. The zero page area is much valued for program and system variables, and, usually, there is no space left for arrays. Also, these addressing modes are particular cases of the more general ABS,X and ABS,Y modes that are what usually get used.

The (IND,X) mode is a clear case of nonsense. It addresses an array of pointers in the zero-page. After writing thousands of assembler-code lines I never found the opportunity of using it. In my opinion it is not only useless, but a classical example of the CISC weakness: A piece of hardware inside the CPU which is rarely used. On the other hand the (IND),Y mode is used very often. In many occasions I missed a similar (IND),X mode. That would have been much more practical.

3.2   Missing useful instructions

The instruction set of the 6502 has few instructions (56), with almost all of them being regularly used in the programs. Due to this some people like to name the 6502 as the "first RISC". The meaning of the term RISC is usually understood to be something more than just a reduced set of instructions. It implies a large set of registers, a load-store architecture and a deep instruction pipeline. None of these characteristics are found in the 6502, but, indeed, its instruction set is reduced, maybe too much. In many occasions the programmers have to resort to tricks, workarounds, or just extra instructions to do simple operations that in other processors are done with single instructions. Some examples of the 6502 instruction set peculiarities follows:

No ADD, SUB

In the 6502 all additions and subtractions include the carry, so, before doing a simple addition you must be sure the carry flag is cleared. This involves another instruction (CLC). The same goes for the subtraction, but in this case the carry has to be set with SEC before executing SBC. I must recognize it is better to have only the addition with carry than having only the addition (this later being a serious flaw for the PIC family of microcontrollers), but, setting the carry before ADC/SUB is a nuisance that makes the code longer and slower. The ADD and SUB instructions would require the ability to force the carry input to the ALU to zero or one, respectively. But this is already done for comparisons, conditional branches and indexed addressing modes, so, the datapath hardware is already there. Only the instruction decoding is missing.

No comparison with carry

While the addition and subtraction always includes the carry, the comparison instruction, CMP, does not. Therefore, when comparing 16 or 32-bit values, the programmer has to resort to the SBC instruction. But that instruction modifies the accumulator. A comparison with carry, CPC, would not have this problem.

No INC A, DEC A

In the 6502 when the accumulator has to be incremented or decremented the ADC or SBC instructions have to be used together with the burden of setting the carry flag properly. This makes the code longer, slower, and the carry flag value is lost. The newer 65C02 includes these instructions at last.

No PHX, PHY, PLX, PLY

In the NMOS 6502 these instructions were missing. They were added later in the CMOS version. They are really useful: Not only they save code and time. They also allows you to preserve the value in the accumulator when saving the X and/or Y registers. As an example consider the following: all registers must be saved before calling a subroutine and then restored. The value in the accumulator has to be preserved for the called routine. We want to do this without modifying any variable in the zero-page or any static allocated memory:

```
6502 code:                              65C02 code:


        pha     ; save registers                pha     ; save registers
        txa                                     phx
        pha                                     phy
        tya                                     jsr     bitbang_out
        pha                                     ply     ; restore registers
        tsx     ; reload Acc from stack         plx
        inx                                     pla
        inx                                     rts     ; return
        inx
        lda     $100,x
        jsr     bitbang_out
        pla     ; restore registers
        tay
        pla
        tax
        pla
        rts     ; return
```

Too few addressing modes for BIT

The BIT instruction only have the ZP and ABS addressing modes. That's a pity because it could be useful for testing the contents of memory without losing the value in the accumulator. With this limitation it is only

useful for testing fixed memory addresses like I/O registers. Again, in the 65C02 there are more addressing modes in general and for the BIT instruction in particular.

No JSR (IND)

The instruction set includes a JMP (IND) but not a JSR (IND). Therefore, when calling a vectorized routine we must call a trampoline routine first:

```
        ...
        jsr        indcall
        ...

indcall:                    ; Trampoline  routine
        jmp        (vector)
```

This approach results in a longer and slower code than doing a JSR (IND). By the way, the JMP (IND) instruction is buggy and care must be taken to ensure your vector does not cross a page [4]. This is not a big problem if variables are properly aligned.

"Volatile" Z and N flags

Almost every instruction modify the Z and N flags, with load instructions being a notable case. This saves instructions for testing data for zero or sign in the code, but, on the other hand, you must do your program bifurcation just after these flags are valid or almost any other code will change them. Other CPUs have fewer flag-changing instructions and you can insert code between comparisons and conditional jumps. This feature also plays a role in the invalid Z flag in decimal mode (see subsection 2.7)

Values contrary to what is normally expected

The carry flag has to be set before SBC or an additional one is subtracted. After SBC the carry flag is set if the result is zero or positive. That is: if the minuend is bigger or equal than the subtrahend. Most other CPUs have the opposite values for their carry flag when doing subtractions, with the carry acting as a "borrow" bit.

The I flag is the IRQ mask. It means that hardware interrupts are masked (or inhibited) when the I flag is set. Therefore, the CLI instruction allows interrupts to happen while SEI disables the interrupts. The CLI mnemonic is found in many other processors with the opposite meaning.

4   Busting myths

There are two basic myths about the 6502 that deserve some analysis. The first is that, as it uses less clock cycles per instruction, the 6502 is faster than other simmilar CPUs. In order to disprove this I choose the Z80 as the CPU to compare the 6502 with. The 6502 and the Z80 have more in common than it is usually though. Both were designed by people who resigned from their former companies after being fed up with their dilbertian bosses. They completed their respectives designs on small companies, contended with their former employers in the market, and won. The 6502 and Z80 wiped the 6800 and 8080 out of the 8-bit market. They were very successful during that era, but both were unable to evolve into competitive 16 or 32 bits designs. The Z80 takes 3 clock cycles to perform a memory read while the 6502 does the same in just one clock cycle. But what this really means is that a memory chip that is just fast enough for a 1 MHz 6502 is equally good for a 3 MHz Z80, and, therefore, Z80 systems were usually running with faster clocks than those based on the 6502. What we have to compare is time, measured in seconds or microseconds, instead of clock cycles.

The other myth I want to address is that of the interrupt latency. It is usually said that the 6502 has a very short interrupt latency, but this doesn't take into account the overhead in the interrupt routine itself. Again, I will compare the 6502 interrupt against that of its main competitor, the Z80.

4.1 6502 vs Z80 speed comparison

During the eighties the 8-bit personal computer market was filled with lots of incompatible computer models. Most of them were powered by the Z80 or 6502 CPUs. Only a small fraction of models relied on other processors, like the 6809. The battle was, thus, served. Z80 and 6502 users were eager to convince each other about the error they made by choosing the opposite CPU. I was also involved in those arguments at that time, and my position was on the Z80 side. Now, many years later, I think I got a more balanced opinion. I'll try to make an objective comparison about the performance of these two processors.

Comparing "apples" and oranges

This is a never ending debate. There are always some pieces of code better suited for any particular CPU architecture and benchmarks tends to be biased. For instance, the Z80 will beat any contemporary CPU when moving data in the memory thanks to its LDIR instruction, but the 6502 can be a winner when doing BCD arithmetic. 6502 fans argue that their CPU uses less clock cycles per instruction than the Z80, but the later usually ran on faster clocks. Also, the Z80 includes more "useful" instructions in its set, meaning that less instructions are required for a particular processing task. So, what we should compare? The benchmark must be neutral in the sense that neither CPU can take advantage of their specific features. But, this also can be considered unfair because a good programmer would use those features whenever possible to get a faster code. It would be a better idea to resort to a real-life application for the benchmark, but, it is not easy to find the same application ported to these very different architectures. And what has to be its source code? A high level language can give different results depending on the compiler or interpreter used. Therefore, what I'm trying to do first is to code a particular application in assembler language for both processors, and I'll try to do my best to reduce the code size and execution time to a minimum in both cases. The application has to be simple, because of the work it would require otherwise, but not too simple or it will not be a representative case. I settled for the following:

A Eratosthenes sieve to compute prime numbers between 2 and 2048. It will require 256 bytes of data to hold a "compressed" sieve and, therefore, single bit, multi-byte addressing is involved. The size of the sieve allows for an efficient addressing in the 6502 case, but it does not matter for the Z80 code. Both processors will have to deal with 16-bit arithmetic, bit manipulation, binary to ASCII conversion and memory filling. I/O can be very system dependent and, while used to test the correctness of the results, will be removed for the final speed test: the dummy character printing routine will only contribute with its call and return delay to the total execution time. When the output is printed the following is obtained:

```
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67  71  73  79  83  89  97  101  103
107  109  113  127  131  137  139  149  151  157  163  167  173  179  181  191  193  197  199
211  223  227  229  233  239  241  251  257  263  269  271  277  281  283  293  307  311  313
317  331  337  347  349  353  359  367  373  379  383  389  397  401  409  419  421  431  433
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
1847  1861  1867  1871  1873  1877  1879  1889  1901  1907  1913  1931  1933  1949  1951
1973  1979  1987  1993  1997  1999  2003  2011  2017  2027  2029  2039
```

The execution time will be measured by executing the code in an emulator and looking at the cycle count. But, what clock frequency should we use for the emulated processors? First, we will take a look to the 8-bit machine survey from Table 1 whose data was mainly collected from Wikipedia [5]. It looks clear that 6502 systems were running at lower clock frequencies, most of them at around 1 MHz, while Z80s were close to 4 MHz. In the following comparison we will use the average frequencies from the table, namely 1.305 MHz for the 6502 and 3.751 MHz for the Z80.

4.1.1 6502 test code

The source code for the 6502 test follows.

```
        *=0

tmp1:                   *=*+1
```

System	CPU	Clock (MHz)	System	CPU	Clock (MHz)
Apple IIe	6502	1.023	ZX Spectrum	Z80	3.5
Commodore 64	6510	1.023	Amstrad CPC	Z80	4
Commodore PET	6502	1	Spectravideo 328	Z80	3.6
Commodore VIC20	6502	1.023	Grundy New Brain	Z80	4
Commodore 128	8502	2	Commodore 128	Z80	4
Atari 2600	6507	1.19	Sony Hit Bit 75 (MSX)	Z80	3.58
Atari 800XL	6510	1.79	VTech Laser 200	Z80	3.57
Oric Atmos	6502	1	Jupiter Ace	Z80	3.25
BBC micro	6502	2	Tatung Einstein	Z80	4
Acorn Atom	6502	1	DEC Rainbow 100	Z80	4.012
Average		1.305	Average		3.751

Table 1: 8-bit system clock rate and CPU survey

```
tmp2 :              *=*+1
number :            *=*+2
index :             *=*+2


        *=$e000



direxe :  ldy      #0                ; Mark all numbers as primes to begin
          lda      #$ff
l1 :      sta      array ,y
          iny
          bne      l1

          sty      number+1          ; start with number=2
          lda      #2
          sta      number

mbuc :    lda      number            ; check if prime
          sta      tmp1
          lda      number+1
          sta      tmp2
          lsr      tmp2              ; y = number /8
          ror      tmp1
          lsr      tmp2
          ror      tmp1
          lsr      tmp2
          ror      tmp1
          ldy      tmp1

          lda      number            ; A = 1<<(number&7)
          and      #7
          tax
          lda      #1
          cpx      #0
          beq      l3
l2 :      asl
          dex
          bne      l2
```

11

```
l3:      and       array ,y           ; check bit
         bne       l35               ; not prime
         jmp       nxn
         ; number is prime. print it
l35:     lda       number
         sta       tmp1
         lda       number+1
         sta       tmp2
         ; tmp1-tmp2: data to be printed
         ldy       #0
prn1:    ;———————————— divide tmp1-tmp2 by 10. Remainder result in A
         ldx       #16
         lda       #0
dv1:     asl       tmp1
         rol       tmp2
         rol
         cmp       #10
         bcc       dv2
         sbc       #10
         inc       tmp1
dv2:     dex
         bne       dv1
         ;————————————
         clc
         adc       #'0'
         pha
         iny
         lda       tmp1
         ora       tmp2
         bne       prn1
         ;————————————
prn2:    pla
         jsr       cout
         dey
         bne       prn2

         lda       #10
         jsr       cout
         ;———————————— Now, mark every multiple of number as not prime
         lda       number            ; index=number
         sta       index
         lda       number+1
         sta       index+1
buc2:    clc                         ; index+=number
         lda       index
         adc       number
         sta       index
         sta       tmp1
         lda       index+1
         adc       number+1
         sta       index+1
         sta       tmp2

         lda       #8                ; if (index>=$800) break
         cmp       index+1
```

12

```
            bcc       nxn

            lsr       tmp2            ; y = index/8
            ror       tmp1
            lsr       tmp2
            ror       tmp1
            lsr       tmp2
            ror       tmp1
            ldy       tmp1

            lda       index           ; A = ~(1<<(number&7))
            and       #7
            tax
            lda       #1
            cpx       #0
            beq       l7
l6 :        asl
            dex
            bne       l6
l7 :        eor       #$ff

            and       array ,y        ; mark the bit
            sta       array ,y
            jmp       buc2


nxn:        inc       number          ; number++
            bne       l5
            inc       number+1
l5 :        lda       number+1        ; if (number&0x7ff)!=0 continue
            cmp       #8
            beq       theend
            jmp       mbuc

theend:     rts


cout :      rts                       ; dummy character output routine

            array=$300        ; 256 byte array
```

This code was run on an emulator modified form the Marat Fayzullin & Alex Krasivsky code. The emulator keep track of the total number of cycles, number of instructions executed and number of dummy fetches.

4.1.2   Z80 test code

The source code for the Z80 test follows.

```
            ; DE: number
            ; HL: index

            org       0x0

            ld        hl ,array       ;Mark all numbers as primes to begin
            ld        de ,array+1
```

```
        ld      bc,255
        ld      a,0xff
        ld      (hl),a
        ldir

        ld      de,2            ;  start with number=2

mbuc:   ld      h,d
        ld      l,e
        srl     h               ;  l=number/8
        rr      l
        srl     h
        rr      l
        srl     h
        rr      l
        ld      bc,array
        add     hl,bc
        ld      a,e             ;  A = 1<<(number&7)
        and     7
        ld      b,a
        ld      a,1
        jr  z,   l2
l1:     sla     a
        djnz    l1
l2:     and     (hl)            ;  check  bit
        jr  z,   nxp

        ;————————————— number  is  prime  —————————
        ;————— print  it  —————

        ld      h,d
        ld      l,e
        ;————————————— divide  HL by  10.  Remainder  result  in  A
        ld      c,0
prn1:   xor     a
        ld      b,16
dv1:    sla     l
        rl      h
        rl      a
        cp      10
        jr  c,   dv2
        sub     10
        inc     l
dv2:    djnz    dv1
        add     a,0x30          ;  convert  to  ASCII  digit
        push    af
        inc     c
        ld      a,h
        or      l
        jr  nz,  prn1
        ld      b,c
prn2:   pop     af
        call    cout
        djnz    prn2
```

```
            ld        a,32      ; space
            call      cout
            ;—————————————— Now, mark every multiple of number as not prime
            ld        h,d
            ld        l,e
buc2:       add       hl,de
            ;—————— if (HL>=$800) break
            ld        a,h
            cp        0x8
            jr        nc,nxp

            push      hl
            ld        a,l
            srl       h                   ; l=index/8
            rr        l
            srl       h
            rr        l
            srl       h
            rr        l
            ld        bc,array
            add       hl,bc
            and       7
            ld        b,a
            ld        a,1
            jr z,     l4
l3:         sla       a
            djnz      l3
l4:         cpl
            and       (hl)
            ld        (hl),a              ; mark the bit
            pop       hl
            jr        buc2

            ;—————————— not prime ——————————
nxp:        inc       de        ; number++
            ld        a,d
            cp        8
            jp nz,    mbuc      ; if (number<2048) continue

            halt

cout:
            ret


array:
```

This code was run in an emulator modified from the Marcel de Kogel code. It keeps track on the total number of cycles and instructions executed. The repeat instructions (LDIR, among others) are recorded as one instruction per each repetition. These instructions can be best considered as single-instruction loops, and, by the way, they can be interrupted.

4.1.3 Compiled code

In order to compare the performance of both CPUs running compiled code the same sieve program was coded using C. The source listing follows:

```
void cout(unsigned char);

void simputch(u8 d)
{
//      cout(d);
}


u8 array[256];
u8 buf[4];

main()
{
    u16 number,index;
    u8 t;
    array[0]=0xff;
    for (t=1;t;t++) array[t]=0xff;

    for (number=2;number<2048;number++) {
        if (array[number>>3]&(1<<(number&7))) {
            //print number
            index=number;
            t=0;
            do {
                buf[t++]=(index%10)+'0';
                index/=10;
            } while (index);
            do {
                simputch(buf[--t]);
            } while (t);
            simputch(' ');

            for (index=number+number;index<2048;index+=number) {
                array[index>>3]&=~(1<<(index&7));
            }
        }
    }
}
```

This code was cross-compiled using "cc65" for the 6502 case and "sdcc -mz80" for the Z80 case. We must be aware that, when comparing the execution times, we are actualy comparing the preformances of both the CPUs and the compilers. The results are discused in the following subsection.

4.1.4 Comparing results

The results of the previous tests are summarized in table 2. These results apply only for the codes presented here and can vary substantially for other applications. Starting with assembler language results: if we can consider these tests as representatives for the average applications we can conclude that the Z80 requires almost double the number of clock cycles than the 6502 to perform the same task, but, when comparing the number of instructions the Z80 only requires about 80 % of the number of instructions of the 6502. This later can be though as the Z80 being "more CISC": the savings come mainly from LDIR and 16-bit arithmetic. At the end, the average Z80 is about a 30 % faster than the average 6502 thanks to its more than double clock rate.

The code size also gets reduced by about a 30% in the Z80 case thanks to its more complex instructions. This can be an interesting result if memory is a concern.

Another interesting result is the percentage of dummy fetch cycles for the 6502. Even with many single byte instructions in the code, the dummy fetches are only an 8.24 % of the total clock cycles. An enhanced

	Assembler			Compiled C		
	6502	Z80	Z80/6502	cc65	sdcc	Z80/6502
Code size (bytes)	202	144	0.71	819	644	0.79
Number of clock cycles	1181744	2282458	1.93	5815906	8220928	1.41
Number of instructions	389087	304526	0.78	1682544	928211	0.55
Number of dummy fetches	97368	—	—	598220	—	—
Clock Frequency (MHz)	1.305	3.571	2.74	1.305	3.571	2.74
Execution time (ms)	905	639	0.706	4457	2302	0.516
Average cycles per instruction	3.04	7.49	2.46	3.46	8.86	2.56
Percentage of dummy fetches	8.24 %	—	—	10.3%	—	—

Table 2: Summary of test results

sequencer with no dummy fetches will only improve the speed of the processor by a 9 % while, probably, requiring many more transistors. Thus, the gain will not worth the effort. A wise design trade-off.

When the compiled codes are compared the Z80/sdcc combination is a clear winner, requiring about half the number of instructions and execution time than the 6502/cc65. That comes at no surprise because the cc65 code includes lots of subroutine calls to library helper functions. This is the result of being compiling code for a CPU that was designed without any regard for high-level languages. In both cases the execution time is much longer than the required for the corresponding assembler language program: about 5 times longer for the cc65 case and 3.6 times larger for the sdcc case.

The average number of clocks per instruction also increases for the compiled codes. This is due to the extensive use of the (ind),y addressing mode and jsr/rts instructions in the 6502 case, while, for the Z80, the use of the registers IX and IY also results in more average clocks per instruction.

As a conclusion: A 1 MHz 6502 has more or less the same processing power as a 2 MHz Z80 when programming the applications directly in assembler language. The performance drops more in the 6502 case when moving to a compiled language due to its inadeccuate instruction set. During the 8-bit computer era most of the Z80 were running with 3.5 to 4 MHz clocks while many 6502 were only 1 MHz, so, on average, the Z80 systems were noticeably faster. What saves the 6502 is the fact that it includes about one half of the transistors of the Z80 and that made it cheaper in a era when CPU chips were really expensive.

4.2   Interrupt latency

The 6502 has a reputation of being very fast at servicing interrupts, its interrupt latency being very short. Some people even claims it is shorter than that of modern CPUs like ARMs! So, lets compare it against its main contender, the Z80. Nobody claims the Z80 is very fast at servicing interrupts, but lets see. In the 6502 processor the interrupt is executed as a modified BRK instruction. This instruction takes 7 clock cycles to execute, and, thus, some people say the interrupt latency is 7 clock cycles. Of course, this simplistic analysis overlooks lots of things. The interrupt latency can be defined as the maximum time lapse between the assertion of the IRQ input and the execution of the related service code. This includes:

1. The time needed for the current instruction to complete its execution. The instruction with the maximum number of cycles has to be considered for a worst case scenario. In the 6502 this instruction is 6 cycles long (after excluding the BRK instruction that will be skipped due to the famous 6502 bug). But there is still another particular case: If the instruction interrupted is a conditional branch the interrupt will be delayed until the next instruction. This adds another 3 cycles for the taken branch. In the Z80 case the longest instruction takes 23 clock cycles.

2. The time needed for jumping to the interrupt service routine. This is 7 cycles for the 6502 and 19 cycles for the Z80 (in interrupt mode 2).

3. The time expended saving the used registers. After the interrupt all the CPU registers must retain their original values in order to not disturb the interrupted program. The interrupt call itself usually only saves the PC and flags. Any other register has to be saved explicitly. On some CPUs there are alternate register sets available for the interrupt routines that avoid saving registers to RAM. This is the case of the Z80: using the EXX and EX AF,AF' instructions there is no need to push anything

into the stack. The 6502 must save the needed registers on the stack, and this usually means the A, X and Y registers, and remember: the NMOS 6502 lacks the PHX and PHY instructions.

4. The time expended investigating the cause of the interrupt and jumping to its particular service code. This is needed when an interrupt is shared between various sources. This is always the case for the 6502, even for a single interrupt source, because the interrupt routine has to discern between a hardware interrupt and the execution of a BRK instruction. On the other hand, the processors with vectorized interrupts can jump to the proper routine directly. In the Z80 case this can be accomplished by putting the CPU into the interrupt mode 2 (vectorized).

Lets compare the beginning of the 6502 and Z80 interrupt routines and lets do some cycle counting:

<table>
<tr><td>6502</td><td>Z80</td></tr>
</table>

```
        ;   9 cycles (curr. instr.)
        ; + 7 cycles (IRQ)

        ; saving registers
pha         ; 3 cycles
txa         ; 2 cycles
pha         ; 3 cycles
tya         ; 2 cycles
pha         ; 3 cycles

        ; finding the IRQ source
bit     IOREG   ; 4 cycles
bpl     nothard ; 2 cycles

;——— Actual ISR code

        ...
        ...
        ...

nothard:        ; check other sources
                ; among them BRK
```

```
        ;   23 cycles (curr. instr.)
        ; +19 cycles (IRQ, IM=2)

        ; saving registers
exx             ; 4 cycles
ex af,af'       ; 4 cycles

;——— Actual ISR code

        ...
        ...
        ...
```

```
Total latency: 35 cycles
               26.8 us  @ 1.305 MHz
```

```
Total latency: 50 cycles
               14.0 us  @ 3.571 MHz
```

Again, the real interrupt latency for the average Z80 is about half of that of the 6502. So, it doesn't looks like the 6502 is so fast when servicing interrupts. The 65C02 can use the PHX and PHY instructions saving 4 cycles, but, this doesn't change the picture too much.

5   RISC vs CISC

Another test is presented in order to compare the 6502 processor with a modern RISC processor. First, I though about using an 8-bit AVR for this purpose, but I lacked a tweakable emulator, and the AVR has lots of instructions making the writing of an emulator just too much work. But, I already have the emulator for my own CPU design: the BN16, and it can be easily adapted for these tests. The BN16 is a 16-bit processor and the comparison could look too much in favor of the RISC. But, as we latter see, the 16-bit architecture is more a handicap than an advantage for this particular test, so, I went ahead with it. But, let present the BN16 processor first.
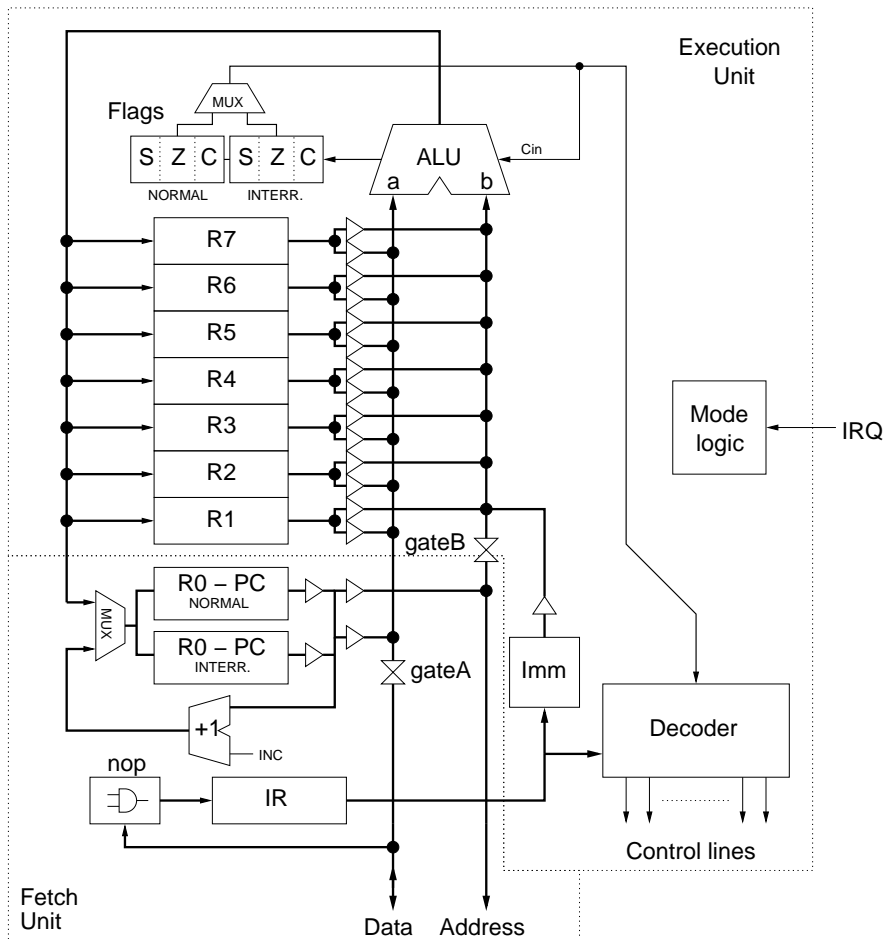
Figure 5: Block diagram of the BN16 CPU

5.1 The BN16 processor

I designed the BN16 processor as a teaching exercise some years ago. It is an extremely simple CPU with a Von Newman memory architecture, a two-stage fetch-and-execute pipeline and only 16 op-codes in its instruction set. Its block diagram is shown in Figure 5. It includes eight 16-bit registers with the R0 register acting as a program counter. Alternative PC and flag registers are used when executing interrupt service routines as there is no formal stack for saving the status of the interrupted program. The instruction register, IR, stores the op-codes read from memory prior to its execution, but, when the memory buses are not available, the IR register is loaded with NOP op-codes. This happens when a load (LD) or store (ST) instruction is executed, turning these instructions into effective 2-cycle instructions. The rest of the op-codes operate with data on the registers and they are executed at a rate of 1 instruction per clock cycle. For the LD and ST instructions only the indexed addressing mode is supported: the memory address is stored in one of the 8 registers of the bank.

The BN16 CPU has a very short instruction set. Its encoding is shown in figure 6. Op-codes are 16-bit long, but only the 4 most significative bits determine the instruction, the rest being the condition codes and operands. All instructions can be conditionally executed depending on the values of the flags and on the bits of their condition codes. A 000 value in the condition code field of any instruction makes it a NOP. Many instructions include three operands: two source registers an a destination register. This makes the design of the decoder easy, but limits the number of available registers to 8.

The BN16 CPU has very few instructions but many of them can operate on the program counter. This opens many possibilities to programmers, like calling subroutines by first copying the PC to other register and then jumping. The PC can also be used as an index register for the LD instruction, thus, providing an immediate addressing mode. The ADDQ and SUBQ instructions are a convenient way of incrementing and decrementing registers, while the JR instruction is also a convenient way to jump to other program location. These instructions have immediate operands encoded in the op-code, and, in the JR case, the program counter is implied as the destination register. This later instruction complicated the decoder design, but it is

Instruction format

3 operands	op	cc	Rd	Ra	Rb
ADDQ,SUBQ	op	cc	Rd	Ra	n
2 operands	op	cc	Rd	xxx	Rb
LD	op	cc	Rd	xxx	Rb
ST	op	cc	xxx	Ra	Rb
JR	op	cc	offset		

op	Instruction		flags		
0000	ADD	Ra,Rb,Rd	C	Z	S
0001	SUB	Ra,Rb,Rd	C	Z	S
0010	ADC	Ra,Rb,Rd	C	Z	S
0011	SBB	Ra,Rb,Rd	C	Z	S
0100	ADDQ	Ra,n,Rd	C	Z	S
0101	SUBQ	Ra,n,Rd	C	Z	S
0110	NOT	Rb,Rd	–	Z	S
0111	NEG	Rb,Rd	–	Z	S
1000	AND	Ra,Rb,Rd	–	Z	S
1001	OR	Ra,Rb,Rd	–	Z	S
1010	XOR	Ra,Rb,Rd	–	Z	S
1011	ROR	Rb,Rd	C	Z	S
1100	LD	(Rb),Rd	–	Z	S
1101	ST	Ra,(Rb)	–	–	–
1110	JR	offset	–	–	–
1111	RETI		–	–	–

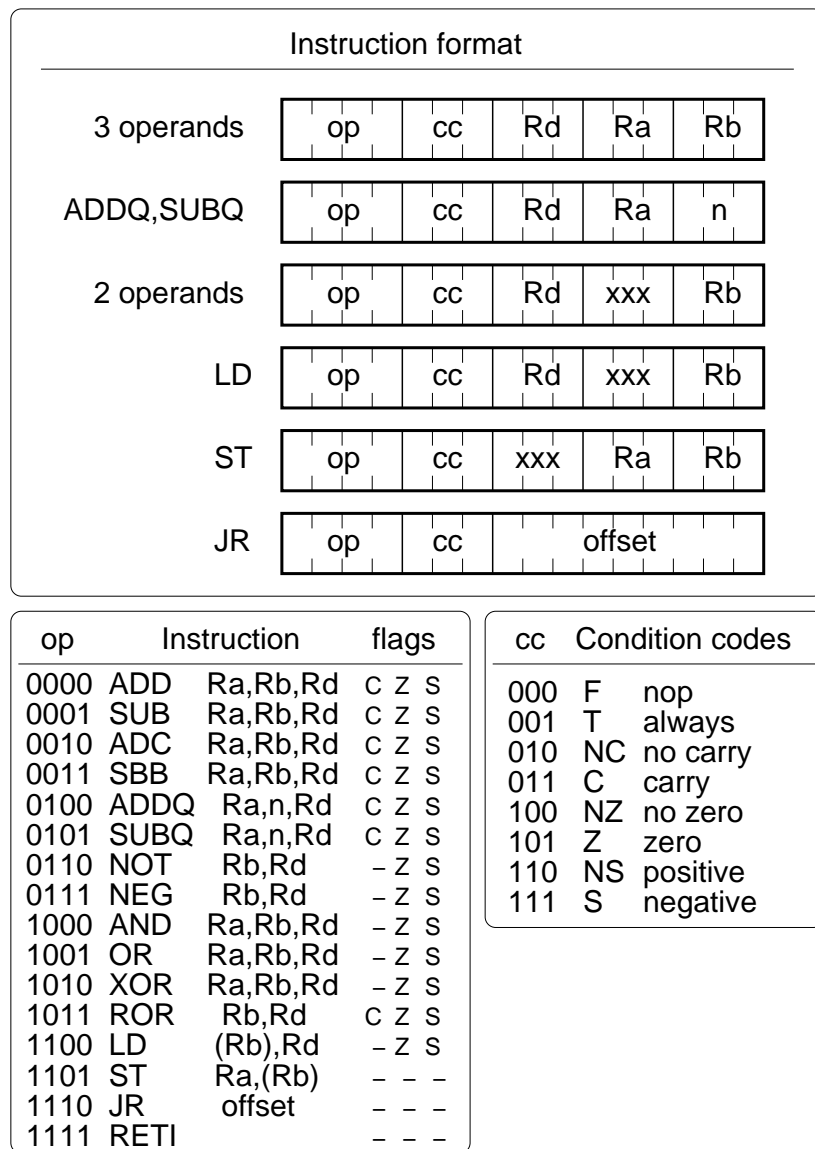| cc	Condition codes	
000	F	nop
001	T	always
010	NC	no carry
011	C	carry
100	NZ	no zero
101	Z	zero
110	NS	positive
111	S	negative

Figure 6: BN16 instruction set encoding

worth the effort. The RETI instruction switches back to the normal PC and flag registers from the interrupt routine.

Due to pipelining the instruction that follows a jump (any instruction with PC as the destination register) is already loaded into the IR register and it will be executed immediately after the jump. The programmer has to place a NOP after a jump if there is no other useful instruction available. This does not apply for the LD (Rx),PC instruction, as the NOP is automatically loaded into the instruction register due to LD.

The BN16 has no stack, but its functionality can be implemented by software. One register has to be used as the stack pointer. Any register would be good, but, for the sake of software compatibility, I choose R1 for SP. Also, another register can be reserved for the PC storage during subroutine calls. I choose R2 for this role and named it LR.

The CPU was designed for a CMOS technology using schematic capture, so, it is detailed to the transistor level. It has an static design, including 7678 transistors. If the CPU were designed as a dynamic, NMOS, chip, the transistor count could be way lower, with a rough estimate being about 3000 to 4000 transistors. This makes the BN16 similar to the 6502 in terms of hardware complexity, and, therefore, justifies even more its choice for the CISC vs RISC test. I was amazed when I learned about the ARM architecture. The BN16 is basically an scaled-down version of the ARM, and I designed it without knowing much about that processor.

5.1.1 BN16 test code

The source code for the BN16 test follows:

```
        org 0     ; RESET vector

        jr      init
        xor     pc,pc,sp        ; Init SP

        org 2     ; IRQ vector

        reti
        nop

init:   ld      (pc),lr         ; Fill sieve with ones
        word    array
        ld      (pc),r3
        word    128             ; 2048/16
        ld      (pc),r4
        word    65535

l1:     st      r4,(lr)
        subq    r3,1,r3
        jr.nz   l1
        addq    lr,1,lr

        ; r7 = number

        ld      (pc),r7         ; start with number 2
        word    2
mbuc:   ror     r7,r6           ; compute pointer
        ror     r6,r6
        ror     r6,r6
        ror     r6,r6
        ld      (pc),lr
        word    4095
        and     r6,lr,r6
        ld      (pc),lr
        word    array
        add     r6,lr,r6
        ld      (pc),r4         ; compute mask
        word    15
        ld      (pc),r3
        word    1
        and     r4,r7,r4
        jr.z    l3
        nop
l2:     subq    r4,1,r4
        jr.nz   l2
        add     r3,r3,r3

l3:     ld      (r6),r4         ; check bit
        and     r4,r3,r3
        jr.z    nextprime
        nop
```

```
                ; Prime number: print it

                ; divide by 10
                xor     r3,r3,r3            ; remainder
                or      r7,r7,r6
                ld      (pc),r5             ; divider
                word    10
                xor     r4,r4,r4            ; number of digits

l35:            ld      (pc),lr             ; loop counter
                word    16

                add     r6,r6,r6            ; shift left
l4:             adc     r3,r3,r3
                sub     r3,r5,r3
                add.c   r3,r5,r3            ; restore value
                addq.nc r6,1,r6            ; update quotient
l6:             subq    lr,1,lr
                jr.nz   l4
                add     r6,r6,r6            ; shift left
                ror     r6,r6              ; undo last shift

                ld      (pc),lr
                word    48                 ; ASCII '0'
                add     lr,r3,r3
                subq    sp,1,sp            ; to stack
                st      r3,(sp)
                addq    r4,1,r4
                or      r6,r6,r6
                jr.nz   l35
                xor     r3,r3,r3

l7:             ld      (sp),r3            ; from stack
                addq    pc,2,lr
                jr      putch              ; print it
                addq    sp,1,sp
                subq    r4,1,r4
                jr.nz   l7
                nop

                ld      (pc),r3
                word    32                 ; space
                addq    pc,2,lr
                jr      putch              ; print it
                nop

                ; Now, mark multiples as not primes
                ; r6 = index
                add     r7,r7,r6           ; index=2*number
bucclr:         ld      (pc).lr            ; compare with 2048
                word    2048
                sub     r6,lr,lr
                jr.nc   nextprime
                nop
                ror     r6,r5              ; compute pointer
```

```
            ror        r5 , r5
            ror        r5 , r5
            ror        r5 , r5
            ld         ( pc ) , lr
            word       4095
            and        r5 , lr , r5
            ld         ( pc ) , lr
            word       array
            add        r5 , lr , r5

            ld         ( pc ) , r4          ; compute mask
            word       15
            ld         ( pc ) , r3
            word       1
            and        r4 , r6 , r4
            jr . z      l9
            nop
l8 :        subq       r4 , 1 , r4
            jr . nz     l8
            add        r3 , r3 , r3


l9 :        ld         ( r5 ) , r4          ; clear bit
            not        r3 , r3
            and        r4 , r3 , r3
            st         r3 , ( r5 )

            jr         bucclr
            add        r7 , r6 , r6          ; add number to index

nextprime :
            ld         ( pc ) , r6          ; compare with max
            word       2047
            sub        r7 , r6 , r6
            jr . nz     mbuc
            addq       r7 , 1 , r7          ; next candidate

hang :      jr         hang
            nop


;————————————————————————————————
; routine to print R3 as an ASCII char

putch :     or         lr , lr , pc          ; return ( dummy print )
            nop

array :     ; The sieve data start here
```

This code differs from that of the 6502 because of the 16-bit data of the BN16 processor. Being able to do 16-bit arithmetic directly means that one instruction is saved for additions or increments, but in the whole listing there are only two cases when this saving is achieved: when adding the number to index and when incrementing number (; next candidate). Comparisons are also 16-bit long, but, in the 6502 case, only the MSBs are compared with the same results, so, there are no savings. On the other hand, when computing the bit mask the 16-bit processor is in a clear disadvantage because it has to iterate more times: The 8-bit mask requires between 0 to 7 shifts, with 3.5 shifts being the average case, while, the 16-bit mask requires 7.5 shifts on average. This means that, not only the shift instruction, but also the related counter decrement and

jump have to be executed twice the times than for an 8-bit CPU, greatly overcoming the savings from the faster addition/increment. Therefore, we must conclude that the Eratosthenes sieve test is biased in favor of the 8-bit CPUs as long as the 16-bit processor is unable to do multibit shifts with single instructions.

The code was run in an emulator (there is not a physical implementation of the BN16 yet), and the cycle count was recorded. Also, the number of forced-NOP fetches and program NOPs were accounted for. The former come from LD or ST instructions, and they help to calculate the average number of cycles per instruction. The program NOPs were included in the code after jumps because a wrong instruction will be executed otherwise.

5.2   Comparison results

	Assembler code test		
	6502	BN16	BN16/6502
Code size	202 bytes	112 words	1.11
Number of instructions	389087	424783	1.09
Number of clock cycles	1181744	472083	0.40
Number of NOPs	—	15616	—
Average cycles per instruction	3.04	1.11	0.36
Percentage of NOPs	—	3.67%	—

Table 3: Summary of test results

The results of the BN16 test and those of the 6502 are summarized in Table 3. Both codes have similar sizes when compared as 2 bytes per word. The number of instructions executed are about a 10% higher for the RISC CPU. This is due to the longer bit mask computation, but also to the fact that RISC instructions are simpler and more are needed to perform the same task. For instance, lets consider a subroutine call. In the 6502 case it is a single instruction, but in the BN16 it is done in three steps:

```
1)   addq    pc,2,lr     ; Save PC+2 (return address) into R2
2)   jr      routine     ; Jump
3)   nop                 ; Executed because of pipelining
```

But, even when executing more instructions, the BN16 processor requires less than half the number of clock cycles than the 6502. By looking at this result it isn't a surprise that CISC CPUs became extinct during the nineties: you can do a lot more with the same number of transistors in a RISC architecture. The key parameter for the high speed of the RISC processor is the low average number of clock cycles per instruction: about 1/3 of the 6502 (and the 6502 was well regarded for its low number of clock cycles per instruction!).

The BN16 also executes many NOPs. If these instructions are subtracted from the total, the average number of cycles per instruction raises to 1.15, that is a 38% of the 6502 case. Still, a huge improvement with respect to the CISC.

Verilog and Spice simulations of the BN16 CPU seems to indicate that it can run with a 30 MHz clock in a 350 nm CMOS technology. The worst case delay comes from the carry propagation in the ALU. It could be improved with carry lookahead circuitry, but, again, the BN16 was designed to be simple. Not because it was intended to be cheap, like the 6502, but because it had to be easy to understand.

References

[1] 6502 Schematic. http://impulzus.sch.bme.hu/6502/download/6502_A4.ps.

[2] Documentation for the NMOS 65xx/85xx Instruction Set. http://nesdev.parodius.com/6502_cpu.txt.

[3] How MOS 6502 Illegal Opcodes really work. http://www.pagetable.com/?p=39.

[4] MOS Technology 6502 (Wikipedia). http://en.wikipedia.org/wiki/6502.

[5] Wikipedia. http://en.wikipedia.org.