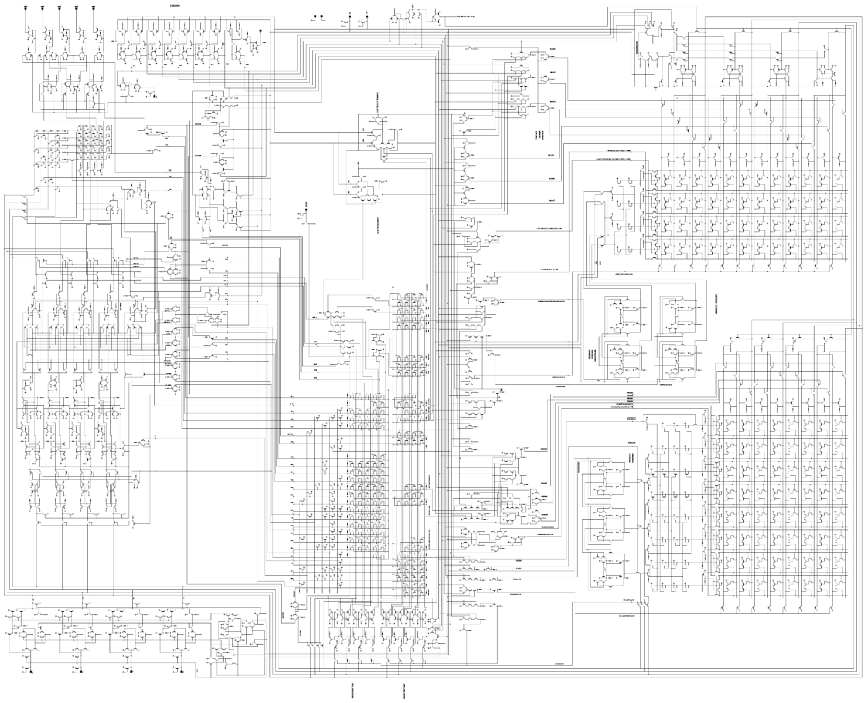
Inside the Intel 4004



Patrick LeBoutillier

Version 1.0, August 2021

Abstract

This paper describes an RTL analysis of the Intel 4004 4-bit CPU. The analysis was achieved by studying a Verilog implementation of the CPU and by using the MCS4 analyzer/simulator. The results are demonstrated using a simulator written in Python.

Introduction

In 2021 will be celebrated the 50th anniversary of the Intel 4004, the world's first commercially produced microprocessor (Wikipedia).

In recent years, Intel has released many documents regarding the 4004 which are relatively easy to find on the Internet (4004.com, DeRamp). Most of these documents fall into 2 distinct categories:

- End-user documentation (user manuals, programming manuals, datasheets, ...)
- Transistor-level schematics

There is therefore very little documentation available on the internals of the 4004 at a level that may be useful for individuals hoping to study the implementation of this CPU, including students and vintage computing enthusiasts.

This is regretful, as the 4004 makes for a great learning platform. It is simple enough to be fully understood by a single individual, yet it implements, in a real-life situation, many of the components one learns about when studying CPUs. As a bonus, the software used in the device for which the 4004 was originally designed, the Busicom 141-PF desktop calculator, is also available, allowing for the study of a production use of the 4004.

The objective of this paper is to provide details on the internal design of the 4004 at a level describing the follow of data between registers, or RTL (Wikipedia). A simulator implementing the described design will also be presented.

Scope

This paper aims to cover the totality of the 4004 microprocessor, with the following exceptions:

- POC (power-on-clear) circuitry
- DRAM precharge or refresh cycle circuitry

Also, the 4004 is implemented using negative logic. For the sake of simplicity, this document will show signals that use positive logic.

Previous Work

The work described in this paper relies heavily on 2 previous works:

- A 4004 analyzer/simulator by Lajos Kintli (Kintli)
    - Lajos Kintli's work validates the schematics released by Intel and allows a simulation of the entire MCS-4 at the transistor-level.
- A Verilog implementation of the 4004 by Reece Pollack (Pollack)
    - Reece Pollack did the tedious work of translating the transistor-level schematics to the higher-level Verilog language, always remaining faithful to the original hardware.

Naming Conventions

In most cases, the names used for the signals will be the ones used in Lajos Kintli's MCS4 analyzer/simulator. Signals may also be inverted with regards to those used in the simulator.

The names used for the registers are the ones used in Reece Pollack's Verilog implementation except when registers use generated numeric names.

In both cases, this paper may use different names for clarity or uniformity purposes.

Typographical Conventions

The names of registers will be in **bold** and the names of signals will be in *italics*. Signals that relate to timing will be prefixed with '@'.

Block-Level Architecture

Fig. 1 shows the block-level architecture of the 4004 as it will be analyzed in this paper.

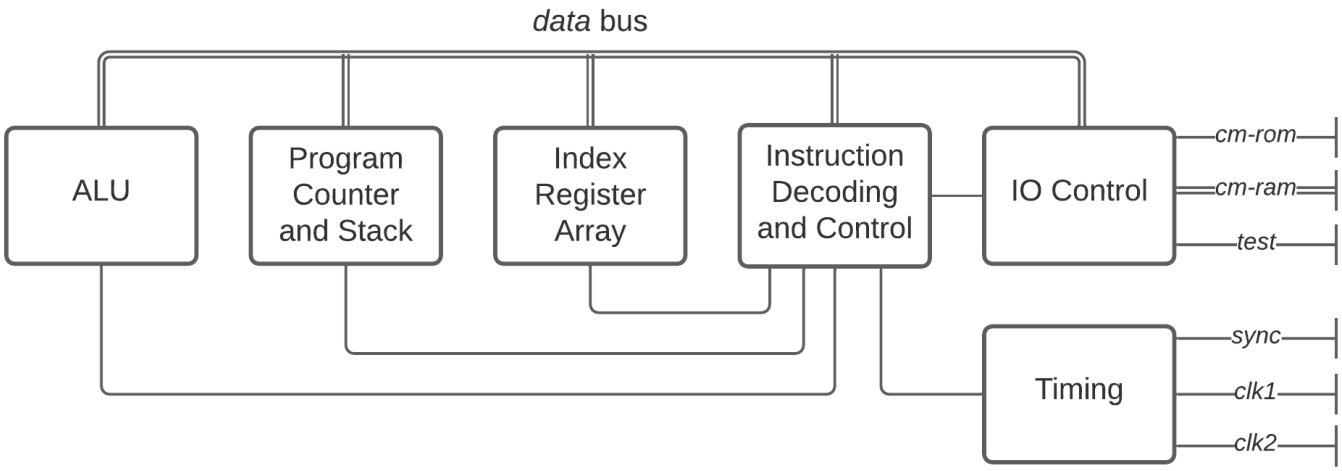


Fig. 1. 4004 Block-level architecture diagram

Note: The interconnections between each block are too numerous to display here, so the Instruction Decoding block is depicted here as a kind of central hub that distributes all the signals to the different blocks. This is not the case in reality.

The remainder of this paper will focus on a detailed analysis of each of these architectural blocks, followed by an introduction of the simulator that implements this RTL analysis.

Timing

Timing in the 4004 is driven by the external two-phased clock. From that are derived numerous signals that guide the instruction cycle through it's 8 possible states. Fig. 2 shows the important signals that orchestrate the operation of instructions in the CPU.

Primary Storage Components

state

The **state** register (8 bits) is a shift register that rotates through the 8 states of instruction execution: A1, A2, A3, M1, M2, X1, X2, X3. It changes to the next state *@clk1*.

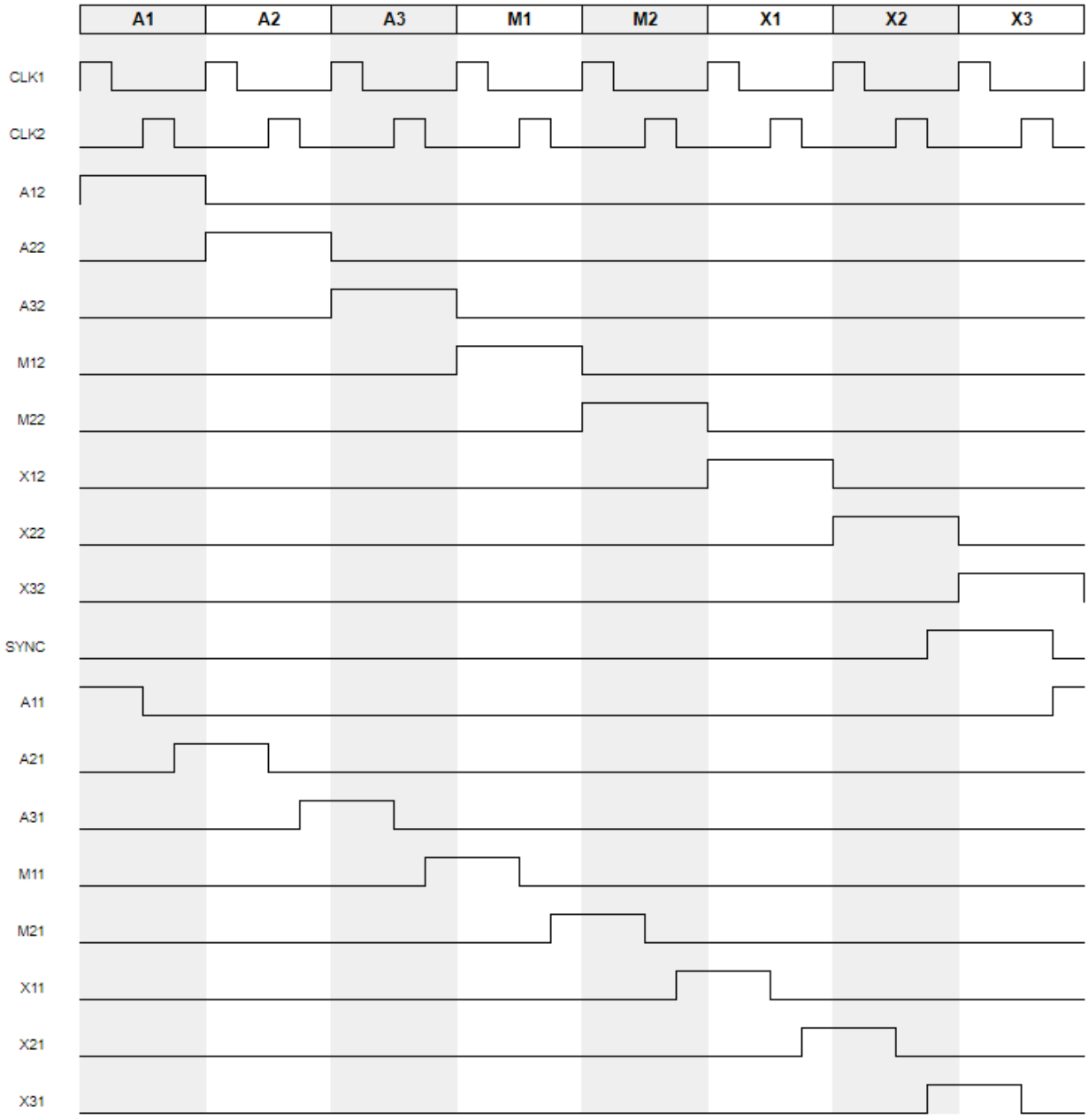


Fig. 2. 4004 Timing signals

Primary Signals

clk1, clk2

clk1 and *clk2* are the two non-overlapping clock signals that are provided to the 4004 from an external source.

a12, a22, a32, m12, m23, x12, x22, x32

These signals correspond to the 8 possible states of the instruction cycle. They are connected to the bits 0-7 of the **state** register.

sync

sync is the synchronisation signal generated by the CPU for the other chips (4001 or 4002) in the system. It is sent out during the last state of the instruction cycle.

Secondary Signals

a11, a21, a31, m11, m21, x11, x21, x31

These signals are generated at many places throughout the CPU. In this paper we will always use these names when these signals are required.

a12clk1, a12clk2, a22clk1, a22clk2, a32clk1, a32ckl2, m12clk1, m12clk2, m22clk1, m22clk2, x12clk1, x12clk2, x22clk1, x22clk2, x32clk1, x32xlk2

These signals are used to subdivide each instruction state into its various phases.

Instruction Decoding and Control

Primary Storage Components

opr, opa

The instruction register is made up of the **opr** (operation code) and **opa** (operand) 4-bit registers.

sc, cond

sc is the single-cycle register. It is used to designate the second instruction cycle for instructions that have 2 cycles (FIN, FIM, JCN, ISZ, JUN, JMS).

cond is the condition register used for instructions that can jump conditionally (JCN and ISZ).

Primary Signals

The primary output signals of the instruction decoding and control block are derived directly from the values of **opr** and **opa**. These signals are used by the rest of the CPU to determine which type of instruction is currently executing. Signals are also derived for specific instruction groups or subgroups when many instructions share the same operations. For example, all the *iow* instructions, which are WRM, WMP, WRR, WR0, WR1, WR2 and WR3, have the same operations in the 4004, since the different behaviours are in this case being performed by the 4002 chip that will process the value sent by the CPU.

The following 3 tables (Table 1, Table 2, Table 3) show the signals that are derived for each instruction. These tables were taken in part from the “Intel 4004 Microprocessor” website (Szyc).

Machine Instructions Signals

Signals			Code	Description	Machine Code		Modifiers
Group	Subgroup	Inst.			1st Inst.	2nd Inst.	
			NOP	No Operation	00000000	-	none
		jcn	JCN	Jump Conditional	0001CCCC	AAAAAAAA	condition, address
	fim+src		FIM	Fetch Immediate	0010RRR0	DDDDDDDD	register pair, data
		src	SRC	Send Register Control	0010RRR1	-	register pair
	fin+jin		FIN	Fetch Indirect	0011RRR0	-	register pair
			JIN	Jump Indirect	0011RRR1	-	register pair
			JUN	Jump Unconditional	0100AAAA	AAAAAAAA	address
		jms	JMS	Jump to Subroutine	0101AAAA	AAAAAAAA	address
			INC	Increment	0110RRRR	-	register
		isz	ISZ	Increment and Skip	0111RRRR	AAAAAAAA	register, address
		add	ADD	Add	1000RRRR	-	register
		sub	SUB	Subtract	1001RRRR	-	register
		ld	LD	Load	1010RRRR	-	register
		xch	XCH	Exchange	1011RRRR	-	register
		bbl	BBL	Branch Back and Load	1100DDDD	-	data
		ldm	LDM	Load Immediate	1101DDDD	-	data

Table 1. Machine instruction signals

IO Instruction Signals

Signals			Code	Description	Machine Code		Modifiers
Group	Subgroup	Inst.			1st Inst.	2nd Inst.	
io	*iow*		WRM	Write Main Memory	11100000	-	none
	iow		WMP	Write RAM Port	11100001	-	none
	iow		WRR	Write ROM Port	11100010	-	none
	iow		WR0	Write Status Char 0	11100100	-	none

	iow		WR1	Write Status Char 1	11100101	-	none
	iow		WR2	Write Status Char 2	11100110	-	none
	iow		WR3	Write Status Char 3	11100111	-	none
	ior	*sbm*	SBM	Subtract Main Memory	11101000	-	none
	ior		RDM	Read Main Memory	11101001	-	none
	ior		RDR	Read ROM Port	11101010	-	none
	ior	*adm*	ADM	Add Main Memory	11101011	-	none
	ior		RD0	Read Status Char 0	11101100	-	none
	ior		RD1	Read Status Char 1	11101101	-	none
	ior		RD2	Read Status Char 2	11101110	-	none
	ior		RD3	Read Status Char 3	11101111	-	none

Table 2. IO instruction signals

Accumulator Instruction Signals

Signals			Code	Description	Machine Code		Modifiers
Group	Subgroup	Inst.			1st Inst.	2nd Inst.	
ope		*clb*	CLB	Clear Both	11110000	-	none
		clc	CLC	Clear Carry	11110001	-	none
		iac	IAC	Increment Accumulator	11110010	-	none
		cmc	CMC	Complement Carry	11110011	-	none
		cma	CMA	Complement	11110100	-	none
		ral	RAL	Rotate Left	11110101	-	none
		rar	RAR	Rotate Right	11110110	-	none
		tcc	TCC	Transfer Carry and Clear	11110111	-	none
		dac	DAC	Decrement Accumulator	11111000	-	none
		tcs	TCS	Transfer Carry Subtract	11111001	-	none
		stc	STC	Set Carry	11111010	-	none
		daa	DAA	Decimal Adjust	11111011	-	none

				Accumulator			
		kbp	KBP	Keyboard Process	11111100	-	none
		dcl	DCL	Designate Command Line	11111101	-	none

Table 3. Accumulator instruction signals

Secondary Signals

More signals are generated by the instruction processing and control block to group together instruction that require similar operations:

Signal	Description	Formula
opa_odd		opa[3]
opa_even		~opa[3]
fin_fim	FIN or FIM	fin_fim_src_jin & ~opa[0]
jcn_icz	JCN or ISZ	jcn_isz
jun_jms	JUN or JMS	jun_jms
ldm_bbl	LDM or BBL	ldm | bbl
inc_isz	INC or ISZ (first cycle only)	(inc | isz) & sc
inc_isz_xch	Instructions that write to a single register	inc | isz | xch
inc_isz_add_sub_xch_ld	Instructions that read from to a single register	inc | isz | add | sub | xch | ld
fin_fim_src_jin	Instruction that read or write to a register pair	fim_src | jin_fin
write_acc	Instructions that save a value in the accumulator	kbp | tcs | daa | xch | cma | tcc | dac | iac | clb | ior | ld | sub | add | ldm_bbl
write_carry	Instructions that save a value in the carry	tcs | tcc | stc | cmc | dac | iac | clc | clb | sbm | adm | sub | add
read_acc	Instruction that use the value of the accumulator as input	daa | rar | ral | dac | iac | sbm | adm | sub | add
add_group	Instructions that perform addition	tcs | tcc | adm | add
inc_group	Instruction that increment	inc_isz | stc | iac
sub_group	Instruction that perform subtraction	cmc | sbm | sub

com	Communication with other chips	See below
cn	Condition flag	See below

Table 4. Secondary signals

Instruction Decoding and Control Timing

Setting opa and opr

opr is set from the *data* bus *@m12clk2* when **sc** is high. *opa* is set from the *data* bus *@m22clk2* when **sc** is high.

Setting sc

For each of the double-cycle instructions (FIN, FIM, JCN, ISZ, JUN, JMS), it is set to 0 *@a12* of the second cycle, or else it is set to 1.

Setting cond

The **cond** register is set to 1 at *@a12* when one of the following conditions is true, or else it is set to 0:

- A JCN instruction is underway (first cycle) and the jump condition (*opa*) matches the current state of the cpu:
  - **sc** & *jcn* & ((~*invert* & *jump*) | (*invert* & ~*jump*))
    - *invert* = *opa*[3]
    - *jump* = (*opa*[2] & **acc_out**) | (*opa*[1] & **cy_out**) | (opa[0] & *test*)
- An ISZ instruction is underway (first cycle) and the output of the adder is not 0
  - **sc** & *isz* & ~*add_0*

In both these cases the conditional jump will occur.

Outputting opa

Some instructions have an argument in *opa* that must the placed on the *data* bus when required:

- If *ldm* or *bbl* or *jun* or *jms* is high:
  - Place *opa* on the *data* bus *@x21*

Program Counter and Stack

The program counter and stack block is responsible for keeping track of addresses in the CPU. The stack is used to store return addresses for subroutine jumps. A stack pointer points to the current stack frame. The program counter contains that address of the instruction that will be fetched next. Fig. 3 shows the program counter and stack block in detail.

This block also contains a special address incrementer that increments the program counter “on the fly” as it is being sent to the data bus during A1, A2 and A3.

Primary Storage Components

sp, row_num, ph, pm, pl, stack0_h, stack0_m, stack0_l, stack1_h, stack1_m, stack1_l, stack2_h, stack2_m, stack2_l, stack3_h, stack3_m, stack3_l

The stack pointer, **sp**, is a 2-bit counter that is used to point to the current frame in the stack. Since **sp** can be modified during the execution of an instruction, a second working register is actually used to commit the value and point into the stack, **row_num**.

ph, **pm** and **pl** comprise the program counter that contains the current working address. The stack consists of 4 12-bit registers, implemented here as 4 x 3 4-bit registers (**stack0_h** through **stack3_l**).

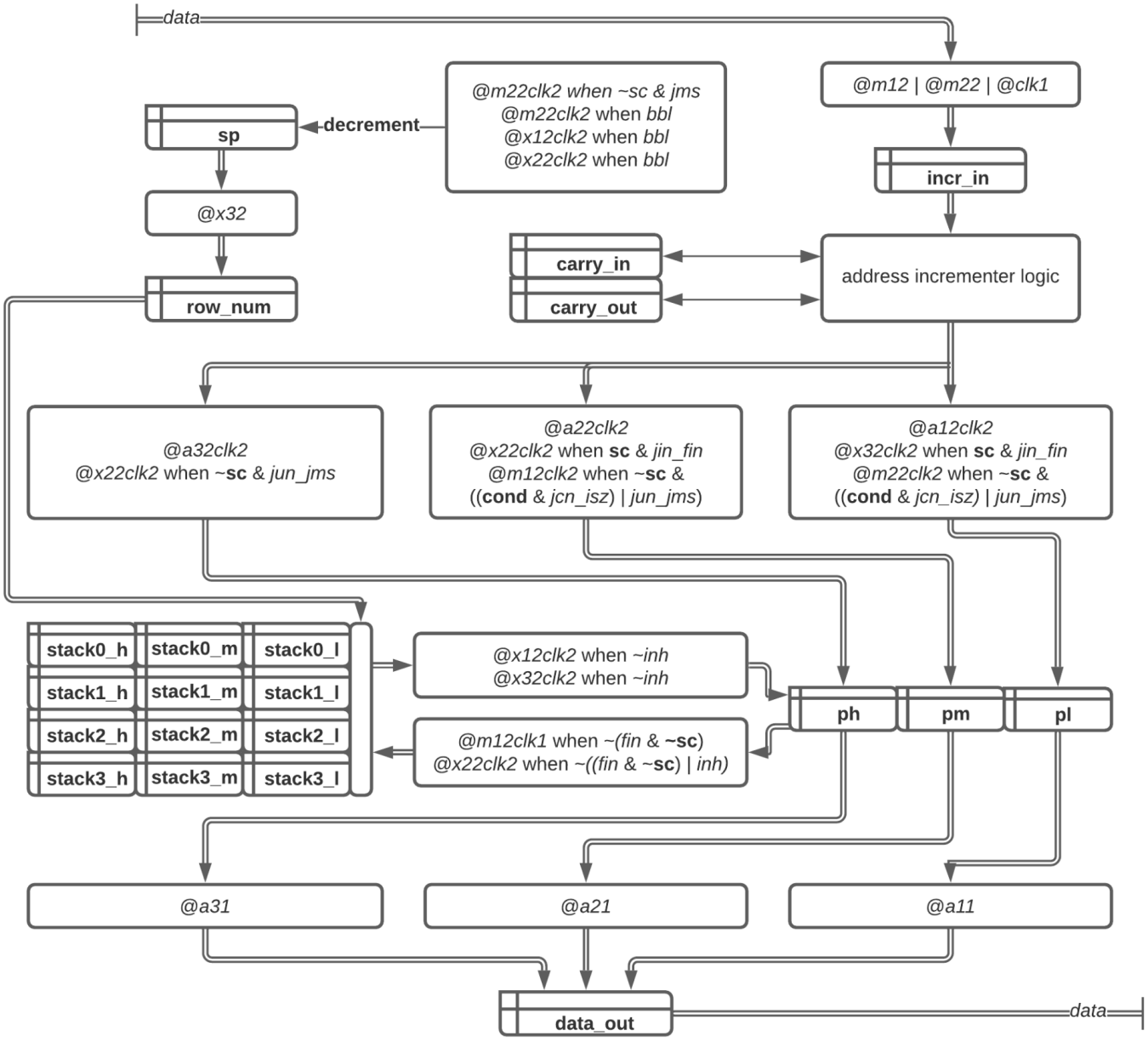


Fig. 3. Detailed program counter and stack diagram

Secondary Storage Components

incr_in, carry_in, carry_out

incr_in is a buffer used to store the values sampled from the bus. It is the input to the address incrementer. **carry_in** and **carry_out** hold the carry values for the address incrementer.

Primary Signals

incr_out

The value coming out of the address incrementer.

inh

inh (inhibit) prevents the commit/refresh process of the program counter during specific instructions. It is defined as such: (jin_fin & sc) | ((jun_jms | (jcn_isz & cond)) & ~sc). The instructions, during those specific cycles, are in the process of modifying the program counter.

Program Counter and Stack Timing

Updating the Stack Pointer

The stack pointer is updated during JMS and BBL under the following conditions:

- If *jms* is high & **sc** is low:
    - @m22clk2, decrement **sp** in order to save the return address on the stack
- If *bbl* is high:
    - @m22clk2, @x12clk2, @x22clk2, increment **sp** in order to return to the previous stack frame.
        - Note: The counter only goes down, so to increment it it must be decremented 3 times.

Sampling the *data* bus

The data bus is sampled when one of the following signals is high: *@m12*, *@m22*, *@clk1*.

- The data is stored in the **incr_in** register
    - It is not initially obvious why sampling the bus *@clk1* is not sufficient to accomplish this task. The reason for this is that for JUN, JMS, JCN and ISZ instructions, the data to be sampled comes from one of the 4001 chips in the system during *@m12* and *@m22*. This data may not be immediately available at *@m12clk1* and *@m22clk1*, so this is why the bus must be sampled for the entire state duration.

Incrementing the Program Counter

During “normal” operation (i.e. excluding some jump and fetch instructions), the program counter must be incremented at every instruction cycle so that the next instruction may be fetched. This is done by sampling the 3 parts of the program counter from the data bus as they are being sent to the ROM (4001) chip(s). Here is how this works:

- @a12clk1
    - Sample the data bus to grab the low nibble of the program counter, and place it in **incr_in** (see “Sampling the *data* bus” above)
- @a12clk2

    - Increment the value in **incr_in**, saving the carry (0 or 1) in **carry_out**
    - Set the value (*incr_out*) in the **pl** register
- @a22clk1
    - Sample the data bus to grab the middle nibble of the program counter, and place it in **incr_in** (see "Sampling the *data* bus" above)
    - Copy **carry_out** to **carry_in**
- @a22clk2
    - Add the value of **carry_in** to **incr_in**, saving the carry (0 or 1) in **carry_out**
    - Set the value (*incr_out*) in the **pm** register
- @a32clk1
    - Sample the data bus to grab the high bit of the program counter, and place it in **incr_in** (see "Sampling the *data* bus" above)
    - Copy **carry_out** to **carry_in**
- @a32clk2
    - Add the value of **carry_in** to **incr_in**, saving the carry (0 or 1) in **carry_out**
    - Set the value (*incr_out*) in the **ph** register

Setting the Program Counter

Besides regular incrementing, there are some situations when the program counter must be set explicitly. This can happen during JIN, FIN, JUN, JMS, JCN and ISZ instructions.

- If (*fin* or *jin* is high) and **sc** is high:
    - We want to set the lower 2 nibbles of the program counter to the values that will be placed on the *data* bus by the index register array block at the appropriate times:
        - *@x22clk2*, save *incr_out* to **pm**
        - *@x32clk2*, save *incr_out* to **pl**
- If (*jun* or *jms* is high) and **sc** is low:
    - We want to set the lower 2 nibbles of the program counter to the values that will be placed on the data bus by the ROM (4001) chip(s) at the appropriate times:
        - *@m12clk2,* save *incr_out* to **pm**
        - *@m22clk2*, save *incr_out* to **pl**
    - We also grab the high nibble of the program counter, in this case *opa*, when it is placed in the data bus for this purpose:
        - *@x22xlk2*, save *incr_out* to **ph**
- If (*jcn* or isz is high) and **cond** is high and **sc** is low:
    - This is a conditional jump instruction, and the jump condition is high (jump). We want to set the lower 2 nibbles of the program counter to the values that will be placed on the data bus by the ROM (4001) chip(s) at the appropriate times:
        - *@m12clk2,* save *incr_out* to **pm**
        - *@m22clk2*, save *incr_out* to **pl**

Committing the Program Counter to the Stack

The program counter is saved regularly to the stack so that it will be already there in the case that a JMS instruction is executed. But in some cases, the program counter is set to a value that is not the next instruction to be executed. For example, during a FIN instruction, that address stored in RRR0 is placed in the program counter in order to be sent out to the ROM chips for retrieval, but it is not where the program needs to go next. For this reason, the logic for committing the program counter to the stack has some exceptions:

- *@m12clk,* the program counter is copied to the stack (at location **row_num**), except when:
    - *fin* is high and **sc** is low:
        - At this point, the contents of the program counter is the address from which we wanted to get the data for the FIN instruction, not the actual program counter. We do not want to commit this value to the stack.

- *@x22clk2*, the program counter is copied to the stack (at location **row_num**), except when:
    - *fin* is high and **sc** is low:
        - See above for details.
    - *inh* is high:
        - When *inh* is high, we are in the process of executing an instruction that is manipulating the program counter. We do not want to commit this value to the stack.

Similarly, there is a process that periodically copies the current stack value to the program counter. This happens @x12clk2 and @x32clk2, except when *inh* is high. Note that is the process that will effectively "rollback" the program counter update made by the FIN instruction.

Reading the Program Counter

The program counter is systematically placed on the data bus *@a11* (**pl**), *@a21* (**pm**) and *@a31* (**ph**) in order to send the requested address to the ROM (4001) chip(s).

Index Register Array

The index register array is the "scratch pad" for the CPU. It consists of 16 4-bit registers that can be read from or written to using various instructions. Fig. 4 shows the index register array block in detail.

Primary Storage Components

r0, r1, r2, …, r15, row_num

r0 through **r15** are the 16 registers in the CPU. They are arranged in an 8 x 2 array, and each of the 8 rows contains 2 registers, one even numbered and the other odd numbered. **row_num** is used to store the number of the row in the array that is currently being worked on (i.e. read from or written to). This always corresponds to the first 3 bits of **opa**, so that value is grabbed from the data bus at *@m22clk2*, which is at the same time that it is placed in **opa**.

Secondary Storage Components

data_in, row_even, row_odd, data_out

data_in is a buffer used to store the value from the data bus required for a write operation. **row_even** and **row_odd** are working buffers that contain the contents of the row being worked on. **data_out** is a buffer used to store the data sent to the bus for a read operation.

Index Register Array Timing

There are 2 main operations supported by the index register array: reading and writing.

Reading Operation

Reading from the register array requires the following steps:

- Setting the row number: This happens systematically *@m22clk2* when **sc** is high. At this moment:
  - **row_num** is set from the *data* bus

- Reading the working row: This happens systematically *@a32clk2* and *@x12clk2*. At this moment:
  - **row_even** and **row_odd** are set from the register pair located in row **row_num**
    - There is an exception *@x12clk2* when both sc and fin are high: in this case, **row_even** and **row_odd** are set from the register pair located in row 0 (see FIN instruction).

- Outputting the requested data:
  - If *inc_isz_add_sub_xch_ld* is high (the instruction reads from a single register) and **sc** is high:
    - If *opa_even* is high:
      - The contents of **row_even** are placed on the data bus *@x21*
    - Else:
      - The contents of **row_odd** are placed on the data bus *@x21*
  - If *fin_fim_src_jin* is high (the instruction reads from a register pair) and **sc** is high:
    - **row_even** is placed on the bus *@x21*
    - **row_odd** is placed on the bus *@x31*

Writing Operation

Here are the steps for writing to the register array:

- Setting the row number: This happens systematically *@m22clk2* when **sc** is high. At this moment:
  - **row_num** is set from the *data* bus

- Setting the data to write: The data bus is sampled when one of the following signals is high: *@m12*, *@m22*, *@clk1*.
  - The data is stored in the **data_in** register
    - It is not initially obvious why sampling the bus *@clk1* is not sufficient to accomplish this task. The reason for this is that for FIN and FIM instructions, the data to be sampled comes from one of the 4001 chips in the system during *@m12* and *@m22*. This data may not be immediately available at *@m12clk1* and *@m22clk1*, so this is why the bus must be sampled for the entire state duration.
- Writing the working row:
  - If *inc_isz_xch* is high (the instruction writes to a single register) and **sc** is high:
    - If *opa_even* is high:
      - The contents of **data_in** are placed in **row_even** *@x32clk2*
    - Else:
      - The contents of **data_in** are placed in **row_odd** *@x32clk2*
  - If *fin_fim* is high (the instruction writes to a register pair) and **sc** is low:
    - **data_in** is copied to **row_even** on *@m12clk2*
    - **data_in** is copied to **row_odd** on *@m22clk2*

- Committing the working row: This happens systematically *@a12clk2* and *@m12clk2*. At these moments:
  - The register pair located in row **row_num** is set from **row_even** and **row_odd**

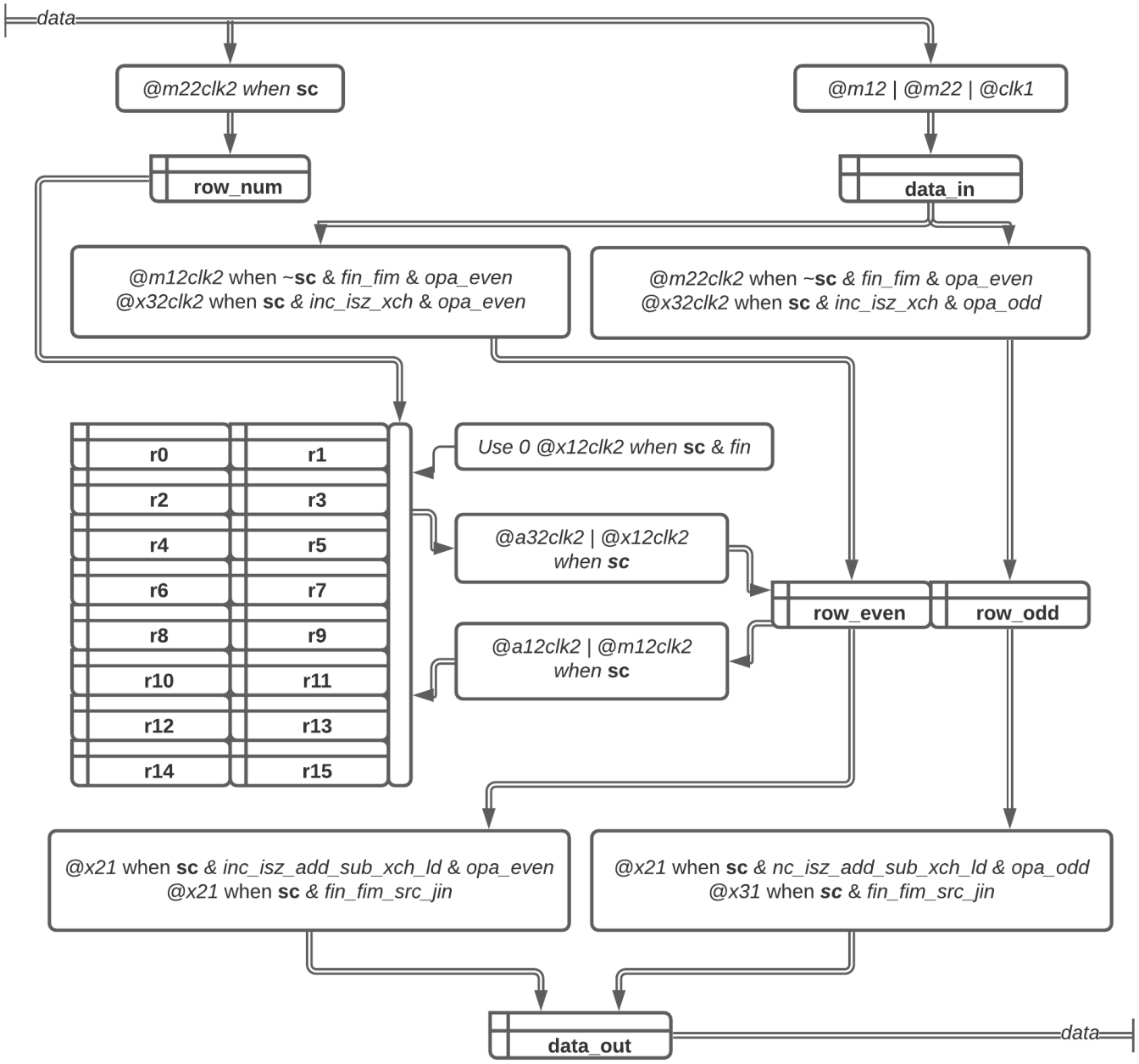


Fig. 4. Detailed index register diagram

ALU (Arithmetic and Logic Unit)

The ALU is where most of the “computation” in the CPU actually takes place. A notable exception is the operation of incrementing the program counter, which has its own incrementing circuit. Fig. 5 shows a detailed diagram of the 4004’s ALU.

Besides storage components and logic for control signals, the ALU contains a 4-bit adder of the carry-lookahead type. There is logic to determine beforehand if a carry may be required for each digit, instead of letting that information “ripple through” the adder. Pollack (Pollack) noted that the adder uses positive logic for bytes 0 and 2, and negative logic for bytes 1 and 3. This is not depicted here.

There is also a shifter right above the accumulator that can shift left or right, using the previous value of the carry as input. Note that the shifter is not available directly from the data bus. To perform a shift operation, the adder must be configured for “pass through” by setting the 2 other inputs to 0.

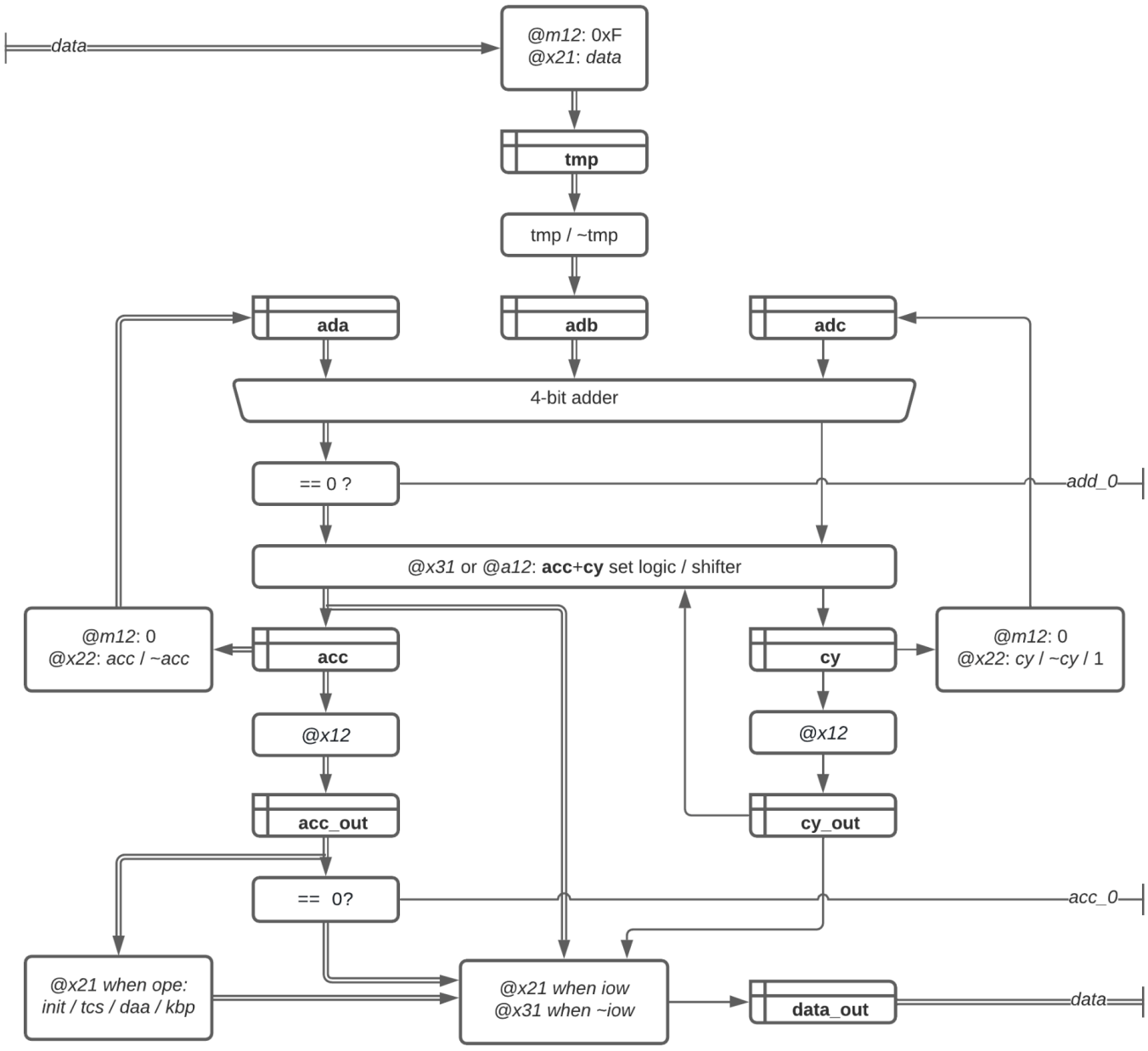


Fig. 5. Detailed ALU diagram

Primary Storage Components

acc, cy

acc (accumulator, 4 bits) and **cy** (carry, 1 bit) are the two main storage registers inside the ALU. They are mainly used to store the result of the adder or of the other operations that can be performed by the ALU (rotate left, rotate right, ...). **acc** is also used as the source for one of the inputs (**ada**) for the adder. **cy** is also used as the source for another input (**adc**) for the adder. It is worth noting that neither of these registers can receive input from the data bus or any source external to the ALU.

tmp

The **tmp** (4 bits) register is connected to the data bus. It is the source for one of the inputs (B) for the adder. The only way to get external data into the ALU is through **tmp**.

Secondary Storage Components

ada, adb, adc

These registers are the actual inputs for the 4-bit adder.

acc_out, cy_out

These registers are set to the current values of **acc** and **cy** during X1. They store the previous value of **acc** and **cy** in case it is needed during an instruction. For example, during an XCH instruction, the value of **acc** is swapped with that of a register from the index, say **r0**. So **acc** can be set to the value of the **r0**, and then the previous value of **acc** can be retrieved from **acc_out** to set **r0**. Also, when **acc** or **cy** must be sent towards the data bus, the values are taken from **acc_out** and **cy_out** respectively.

data_out

data_out is connected to the data bus. Depending on what is required, it can be set to **acc_out**, **cy_out**, *add* (see below) or the results of the *data* bus initialization routine (see below)

Primary Signals

add_0, acc_0

These output signals are flags used elsewhere in the CPU as conditions for jump-type instructions. *add_0* indicates if the contents of **acc** is equal to 0, and *add_0* indicates if the output of the adder (i.e. the sum, *add*) is 0.

Secondary Signals

add, co

We will name *add* the internal signal directly at the output of the adder. Pollack (Pollack) calls this signal *acc_in*. The internal carry signal coming out of the adder will be called *co*.

ALU Timing

In a CPU, the ALU is often thought of as a purely combinational circuit. In the 4004, this is in fact the case for the adder and the shifter, but the use of the ALU as a whole is highly orchestrated.

Using the ALU requires 4 steps that occur during various time signals:

- Initializing the adder inputs (init): The first initialization step happens *@m12*. At this moment:
    - **ada** is set to 0
    - **tmp** is set to 0xF (which ultimately drives **adb**)
    - **adc** is set to 0

A second initialization step (the data bus) happens *@x21* for *ope* instructions:
    - If the 3rd bit of *opa* is low, *data* is set to 0
    - Else:
        - If *daa* is high, *data* is set to 6
        - If *tcs* is high, *data* is set to 9
        - If *kbp* is high, *data* is set according to the value of **acc_out** (see KBP instruction for details)
        - Else *data* is set to 0xF

- Setting the adder inputs (set): This happens *@x22clk1* for *~io* instructions and during *@x22clk2* for *io* instructions. At this moment:
    - **tmp** is set to the value on the *data* bus:
        - If *sub_group* is high for the current instruction, **adb** is set to ~**tmp**
        - Else **adb** is set to **tmp**
    - If *read_acc* is high, **ada** is set to **acc**
    - If *cma* is high, **ada** is set to ~**acc**
    - If *add_group* is high, **adc** is set to **cy**
    - If *sub_group* is high, **adc** is set to ~**cy**
    - If *inc_group* is high, **adc** is set to 1

- Saving the results (save): This happens *@x31* for *~ior* instructions and during *@a12* for *ior* instructions. At this moment:
    - If *ral* is high, {**cy**, **acc**} is set to {*add*, **cy_out**}
    - If *rar* is high, {**acc**, **cy**} is set to {**cy_out**, *add*}
    - If *write_acc* is high, **acc** is set to *add*
    - If *write_carry* is high, **cy** is set to *co*

- Outputting the result (enable):
    - If *iow* is low:
        - If *inc_isz* is high, **data_out** is set to *add* @x31
        - If *xch* is high, **data_out** is set to **acc_out** @x31
    - If *iow* is high:
        - **data_out** is set to **acc_out** @x21
        - **data_out** is set to {3'b000, **cy_out**} **@x31**
            - It is unclear what purpose this serves, as no instruction seems to read this data.

IO Control

Primary Storage Components

ram_bank

The **ram_bank** register is used to store the RAM bank selected by the DCL instruction. This happens *@x21*.

Primary Signals

cm-rom, cm-ram, test

cm-rom is the command signal for the ROM (4001) chips. *cm-ram* is the command signal for the RAM (4002) chips. The *test* signal is external to the CPU and can be tested during a JCN instruction.

IO Control Timing

cm-rom and *cm-ram* are both turned on and off at the same time to indicate that communication with other chips is required:

- *@a31* (off at *@m11*), to signal to the ROM chips that the last nibble of the requested instruction address is on the *data* bus
- *@m21* (off at *@x11*), when *io* is high, to signal to the RAM chips that an IO instruction is being processed
- *@x21* (off at *@x31*), when *src* is high, to signal to the ROM/RAM chips that one is being selected for RAM/IO operation.

Simulator

A simulator for the Busicom 141-PF was written in the Python language to implement the RTL analysis. It runs the actual Busicom code available on the 4004.com site. Fig. 6 shows the original Busicom 141-PF keyboard.

Fig. 6. Busicom 141-PF keyboard

Why Python?

Python was chosen in order to make the code accessible to the largest audience possible. Although Python is relatively slow for this type of project, it is generally easy to read and to understand. Also, the decorator feature in Python makes for a nice way to cleanly assign code blocks to specific timing events.

For example, fig. 7 shows the code the implements the INC instruction:

```
# INC
opr, opa = 0b0110, any
@X21
def _():
    scratch.enableReg()
@X22clk1
def _():
    alu.setADC(one=True)
@X31
def _():
    alu.runAdder()
    alu.enableAdd()
@X32clk2
def _():
    scratch.setReg()
```

Fig. 7. Sample code for INC instruction

Basic Operation

The simulator has no graphical user interface, is it a command-line (CLI) application. When a keystroke is expected, the user is presented with a status line and a prompt. The user enters a command that represents a keystroke from the original calculator and presses ENTER. Any lines that would be printed on the printer are displayed on the screen. Here is a simple example:

```
$ ./141-PF.sh
### DP[0] RND[F] ( )( )( ): 2
### DP[0] RND[F] ( )( )( ): +
>>>  |              2 +     |
### DP[0] RND[F] ( )( )( ): 2
### DP[0] RND[F] ( )( )( ): +
>>>  |              2 +     |
### DP[0] RND[F] ( )( )( ): =
>>>  |              4    *  |
>>>  |                      |
### DP[0] RND[F] ( )( )( ): q
```

Here is a more complicated example using some of the specialized keys and showing multiple keystrokes entered on the same line:

```
$ ./141-PF.sh
### DP[0] RND[F] ( )( )( ): ddrr
### DP[2] RND[T] ( )( )( ): 1#
>>>  |              1    #  |
```

```
                        ### DP[2] RND[T] ( )( )( ): 1+
                        >>>  |           1.00 +     |
                        ### DP[2] RND[T] ( )( )( ): 2+
                        >>>  |           2.00 +     |
                        ### DP[2] RND[T] ( )( )( ): 3+
                        >>>  |           3.00 +     |
                        ### DP[2] RND[T] ( )( )( ): #
                        >>>  |           6.00 <>    |
                        ### DP[2] RND[T] ( )( )( ): 2#
                        >>>  |              2    #  |
                        ### DP[2] RND[T] ( )( )( ): 4+
                        >>>  |           4.00 +     |
                        ### DP[2] RND[T] ( )( )( ): 5+
                        >>>  |           5.00 +     |
                        ### DP[2] RND[T] ( )( )( ): 6+
                        >>>  |           6.00 +     |
                        ### DP[2] RND[T] ( )( )( ): #
                        >>>  |          15.00 <>    |
                        ### DP[2] RND[T] ( )( )( ): 3#
                        >>>  |              3    #  |
                        ### DP[2] RND[T] ( )( )( ): 7+
                        >>>  |           7.00 +     |
                        ### DP[2] RND[T] ( )( )( ): 8+
                        >>>  |           8.00 +     |
                        ### DP[2] RND[T] ( )( )( ): 9+
                        >>>  |           9.00 +     |
                        ### DP[2] RND[T] ( )( )( ): =
                        >>>  |          45.00    *  |
                        >>>  |                       |
                        ### DP[2] RND[T] ( )( )( ): q
```

Here is the help screen that indicates the accepted commands other than numbers and '.':

```
$ ./141-PF.sh
### DP[0] RND[F] ( )( )( ): h

Keyboard (enter a sequence of keys and press enter):

                a     d--------  r--

                S     7  8  9    -  #         CM
                EX    4  5  6    +  /    %    RM
                CE    1  2  3       *    M+   M-
                CL    0     .     =      M=+  M=-

a:  Paper feed button   d:  Decimal point selector   r:   Round off switch
S:  Minus sign                                       CM:  Clear memory
EX: Exchange                                         RM:  Recall memory
```

```
CE: Clear entry         M+:  Memory +               M-:  Memory -
CL: Clear               M=+: Memory equals +        M=-: Memory equals -

Status line:

    ### DP[0] RND[F] ( )( )( ):
           |      |   |  |  |
           |      |   |  |  ----> Memory light
           |      |   |  -------> Negative light
           |      |   ----------> Overflow light
           |      --------------> Rounding mode (F:float, R:round, T:truncate)
           ----------------------> Decimal points (0-8)
```

Conclusion

I believe the work presented here to be a faithful representation of the internal design and timing of the Intel 4004 microprocessor, and that this is proven by the accompanying simulator. Hopefully it will allow vintage CPU enthusiasts to better understand the internals of this classic microprocessor.

Future Work

This work could be completed by analyzing the POC, reset and refresh circuits used in the chips. Also, equivalent works could be produced concerning the other chips in the MCS-4 family, namely the 4001, 4002 and 4003 chips. This would allow a complete RTL documentation of the entire MCS-4 system. Finally, a C-language version on the simulator could be written to provide better performance.

Acknowledgements

Many thanks to Tim McNerney for giving me the initial information that allowed me to get started in this paper. I also thank Lajos Kintli and Reece Pollack for doing the groundwork upon which the contents of the paper is built.

Appendix

GitHub Repository

All the code and documentation collected during this project is collected in the following Github repository:

- https://github.com/patrickleboutillier/inside4004/

Detailed 4004 Timing

A master table showing the detailed timing information for the 4004 was produced along with this paper. The latest version can be found here:

- https://github.com/patrickleboutillier/inside4004/blob/master/doc/MCS-4-4004-Detailed-Timing.pdf

Citations and References

DeRamp. “4004 Family” *Vintage Computing*,

https://deramp.com/downloads/mfe_archive/050-Component%20Specifications/Intel/Microprocessors%20and%20Support/4004%20Family/

4004.com. “Intel 4004” *Intel 4004*, 2006,

https://www.4004.com/

Kintli, Lajos. “MCS4 analyzer/simulator” *4004.com*, 2018,

https://www.4004.com/assets/i400x_analyzer_20210324.zip

Pollack, Reece. “4004 CPU and MCS-4 family chips” *4004 CPU and MCS-4 family chips*, 2020,

https://opencores.org/projects/mcs-4

Pollack, Reece. “Insanity 4004” *Insanity 4004*, 2020,

http://insanity4004.blogspot.com/

Szyc, Maciej. “Intel 4004 Instruction Set” *Intel 4004 Microprocessor*, 2007,

http://e4004.szyc.org/iset.html

Wikipedia. “Intel 4004” *Wikipedia*,

https://en.wikipedia.org/wiki/Intel_4004

Wikipedia. “Register transfer-level” *Wikipedia,*

https://en.wikipedia.org/wiki/Register-transfer_level