International Journal of Scientific Research and Engineering Development-– Volume 9 Issue 5, Sep-Oct 2026
Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

Arithmetic Without the Accumulator: A Tutorial Analysis of 16-Bit Addition, ALU Data Paths, and PSW Behaviour in the Intel 8085 Microprocessor

Shruti Chouhan[1], Giriraj Kumar Prajapati[2], Neha Khare[3]
[1]M.Tech. Scholar, [2]Head of Department, [3]Assistant Professor
[1,2,3]Department of Computer Science, Alpine Institute of Technology, Ujjain (M.P.), India
shrutichouhan798@gmail.com, prajapatigiri38@gmail.com, nehakhareofficial@gmail.com

Abstract:

The Intel 8085 remains a standard teaching vehicle for computer organization because of its small, accumulator-centric instruction set. Although the device is an 8-bit processor, its instruction set includes 16-bit arithmetic, most notably the DAD instruction, which adds a register pair to HL without naming the accumulator and without disturbing four of the five documented condition flags. This asymmetry is a recurring source of conceptual difficulty: learners who have internalised the rule that "every ALU operation passes through the accumulator" cannot reconcile it with the existence of DAD, and they routinely predict incorrect flag and Program Status Word (PSW) states in examinations and in laboratory work. This tutorial paper resolves the apparent contradiction by separating three distinct arithmetic resources inside the 8085: the accumulator-centric 8-bit data path, the hidden temporary registers (ACT and TMP) that allow the same 8-bit ALU to be driven from a register pair while bypassing the accumulator, and the 16-bit incrementer/decrementer attached to the address latch that serves INX, DCX and program counter updates. We show, using the published instruction timing of DAD (one byte, three machine cycles, ten T-states, of which six are bus-idle), that DAD is executed as two successive 8-bit ALU passes with internal carry propagation rather than by a dedicated 16-bit adder, and we correct a widespread textbook simplification to that effect. The flag and PSW consequences are then derived rather than asserted, and are verified against a worked simulator run. A quantitative comparison shows that the DAD form of a 16-bit addition requires 1 byte and 10 T-states against 6 bytes and 24 T-states for the equivalent MOV/ADD/ADC sequence, while additionally preserving the accumulator and four condition flags. The paper concludes with a misconception-to-remedy table, measurable learning outcomes and ready-to-use assessment items, so that the material can be adopted directly in undergraduate microprocessor courses.

***Keywords*—**Intel 8085, DAD instruction, arithmetic logic unit, data path, Program Status Word, condition flags, address arithmetic, instruction timing, microprocessor education, student misconceptions

I. INTRODUCTION

Addition is the most frequently executed arithmetic operation in any stored-program computer, and the manner in which a processor implements it reveals a great deal about its internal organisation. Early microprocessors were built around narrow data paths, so operations on operands wider than the natural word length had to be synthesised from several narrower operations. The Intel 8085, an 8-bit device, is the canonical classroom example of this design compromise: it processes data eight bits at a time, yet it must manipulate 16-bit addresses continuously in order to fetch instructions, traverse arrays, service the stack and update pointers [1], [2].

Because of this dual requirement, the 8085 instruction set is not uniform. The arithmetic and logical instructions that operate on data—ADD, ADC, SUB, SBB, ANA, ORA, XRA and CMP—are strictly accumulator-centric: one operand is implicitly the accumulator, the result is written back to the accumulator, and all five documented condition flags are updated [3]. Alongside these sits a small group of 16-bit instructions—DAD, INX, DCX, LHLD, SHLD and XCHG—that operate on register pairs and, in the case of DAD, perform a true addition of two 16-bit quantities without mentioning the accumulator at all and without changing the Sign, Zero, Auxiliary Carry or Parity flags.

For a student who has just been taught that the accumulator is mandatory for arithmetic, the DAD instruction reads as a contradiction of the rule. Classroom experience with undergraduate microprocessor courses shows that the difficulty is persistent and produces a family of related errors: learners assert that DAD must secretly use the accumulator, that the Zero flag must be set when HL becomes 0000H, that PUSH PSW after a DAD saves a changed accumulator, or that a 16-bit addition can be tested for a zero result with a JZ instruction. These are not careless slips; they are the predictable consequence of an over-generalised mental model, the kind of

durable misconception that concept-inventory research in other engineering disciplines has documented extensively [4], [5].

The conventional remedy offered in many laboratory manuals and lecture notes is to state that the 8085 contains "a separate internal 16-bit adder" used by DAD. This answer removes the contradiction, but at the cost of accuracy. Die-level reverse engineering of the 8085 shows that the device contains exactly one arithmetic unit, and that it is eight bits wide [6], [7]; the only 16-bit arithmetic circuit on the chip is an incrementer/decrementer attached to the address latch, and that circuit supports program counter stepping, INX and DCX, not DAD [8], [9]; the flag logic that decides which condition bits an instruction is permitted to update has been mapped in the same body of work [10]. The published timing of DAD—ten T-states, of which six are bus-idle—is itself evidence that the operation is carried out as two sequential 8-bit additions rather than in a single wide adder. Replacing one misconception with another is a poor bargain in a tutorial setting, because the "16-bit adder" story cannot explain why DAD costs 10 T-states when ADD costs 4, nor why INX, which genuinely does use a 16-bit circuit, affects no documented flag at all.

This paper therefore sets out to give an explanation that is simultaneously accurate at the hardware level and usable in a classroom. Its contributions are as follows:

1) A clear separation of the three arithmetic resources of the 8085—the accumulator-centric 8-bit data path, the accumulator-bypassing path into the same 8-bit ALU through the hidden ACT and TMP registers, and the 16-bit incrementer/decrementer of the address latch—together with the set of instructions served by each.
2) A timing-based argument, using the documented one-byte, three-machine-cycle, ten-T-state execution profile of DAD, that establishes the two-pass 8-bit interpretation without requiring the reader to inspect silicon.
3) A derivation, rather than an assertion, of the flag and PSW behaviour of DAD, including the consequences for PUSH PSW and POP PSW and for conditional branching after a 16-bit addition.
4) A quantitative cost comparison between the DAD form of a 16-bit addition and the equivalent accumulator-based MOV/ADD/ADC sequence, expressed in bytes, T-states, microseconds and side effects.
5) A set of worked and simulator-verified examples, a misconception-to-remedy table, measurable learning outcomes and matching assessment items that instructors can adopt directly.

The remainder of the paper is organised as follows. Section II reviews related tutorial and pedagogical work. Section III describes the architectural background required for the argument. Section IV analyses the accumulator-centric 8-bit arithmetic path and its limitations for address computation. Section V introduces the DAD instruction and its encoding and timing. Section VI resolves the central misconception and presents the corrected data-path model. Section VII derives flag and PSW behaviour. Section VIII gives worked examples and the quantitative comparison, and Section IX reports the simulator verification. Section X collects misconceptions and remedies, Section XI discusses educational deployment, Section XII states limitations and future work, and Section XIII concludes.

II. RELATED WORK AND POSITIONING

Standard textbooks treat the material of this paper in one of two ways. Programming-oriented texts such as Gaonkar [1] and Padmanabhan [11] present DAD as an instruction with a stated effect—HL receives HL plus the named register pair, and only the Carry flag is affected—and move on to applications. The statement is correct and sufficient for writing programs, but it is behavioural rather than explanatory, and it is precisely at the point where a learner asks "why only Carry?" that the account stops. Architecture-oriented texts such as Stallings [12] and Hayes [13] explain data paths, temporary registers and multi-byte arithmetic in general terms, but they do not specialise the discussion to the 8085 instruction set that the student is actually programming.

A third body of material—laboratory manuals, question banks and lecture notes—attempts to bridge the gap with the "separate 16-bit adder" formulation criticised in Section I. Advanced treatments of the 8085 and its peripherals [14], [15], [16] describe the incrementer/decrementer associated with the address latch correctly, but the distinction between that circuit and the path used by DAD is rarely drawn sharply, and the two are often conflated in summary tables that list DAD, INX, DCX and PC increment as users of a single "16-bit unit".

The hardware facts needed to settle the question are available from die-level reverse engineering of the 8085 published by Shirriff [6]–[10], which documents an eight-bit-wide ALU, the programmer-invisible ACT and TMP registers that feed it, the control lines that load ACT from a source other than the accumulator for DAD and related instructions, the flag circuits, and the separate 16-bit increment/decrement circuit beneath the register file. That work is written for a hardware

audience and is not framed as instruction-level pedagogy. The contribution of the present paper is to connect these two literatures: to take the architectural evidence, express it in terms of instructions, flags and T-states that an undergraduate already possesses, and package the result as teachable material with worked examples and assessment items.

The pedagogical framing follows established practice in engineering education research. Concept inventories and misconception studies in physics [4] and in introductory programming [5], [17] show that incorrect but internally consistent mental models survive ordinary instruction and must be addressed explicitly by confronting the learner with the case the model fails to explain. Cognitive load theory [18] further suggests that the correct model should be introduced by extending a structure the learner already has—here, the familiar accumulator data path—rather than by presenting an unfamiliar block diagram in full. Both principles shape the order of presentation adopted in Sections IV to VII.

III. ARCHITECTURAL BACKGROUND

A. Registers Visible to the Programmer

The 8085 provides an 8-bit accumulator A, six general-purpose 8-bit registers B, C, D, E, H and L, a 16-bit stack pointer SP and a 16-bit program counter PC. The general-purpose registers are addressable individually or as the pairs BC, DE and HL. The HL pair is privileged: it is the implicit memory pointer for the M operand, the implicit destination of DAD, and the register loaded and stored by LHLD and SHLD [1], [2].

B. Registers Not Visible to the Programmer

Two 8-bit temporary registers, conventionally named ACT (accumulator temporary) and TMP, sit between the register file and the ALU. They do not appear in the programming model and cannot be named in an instruction, but every ALU operation uses them: the ALU takes its two operands from ACT and TMP rather than directly from the accumulator and the addressed register [7]. This detail is usually omitted from introductory teaching, and its omission is the root cause of the misconception this paper addresses, because it is the ability to load ACT from a source other than the accumulator that allows an instruction to perform arithmetic while leaving A untouched.

C. The Arithmetic Logic Unit

The 8085 ALU is eight bits wide and is constructed from eight one-bit slices. It performs addition, AND, OR, XOR and right shift, with complement and subtraction obtained by inverting one input, under the control of a programmable logic array driven by the opcode and by the position within the instruction cycle [6], [7]. There is no second, wider arithmetic unit on the die. Every arithmetic result in the machine, including the two halves of a DAD result, is produced by this one 8-bit circuit.

D. The 16-Bit Incrementer/Decrementer

Beneath the register file the 8085 carries a 16-bit address latch together with an increment/decrement circuit built as a chain of half-adders with ripple carry [9], the standard construction analysed in digital design texts [19], [20]. This circuit steps the program counter after each fetch, adjusts the stack pointer on PUSH, POP, CALL and RET, and implements the INX and DCX instructions. It is an incrementer, not a general adder: it can add or subtract a constant of one (and, as an optimisation for conditional jumps, two), but it cannot add an arbitrary 16-bit operand. This limitation is decisive for the argument of Section VI, because DAD must add an arbitrary register pair and therefore cannot be served by this circuit.

E. The Flag Register and the Program Status Word

The 8085 flag register is eight bits wide. Five flags are documented by Intel and examinable in any course: Carry (CY), Auxiliary Carry (AC), Sign (S), Zero (Z) and Parity (P). Two further bits are implemented on the silicon but were never documented by Intel: the signed overflow flag V in bit 1 and the K (also called X5) flag in bit 5, the latter driven by the carry out of the 16-bit incrementer/decrementer or by the result of a signed comparison [8], [10]. Bit 3 is unused and reads as zero. Table I summarises the layout.

TABLE I. Layout of the 8085 flag register

Bit	Symbol	Name	Status	Meaning and source
7	S	Sign	Documented	Copy of bit 7 of the 8-bit ALU result
6	Z	Zero	Documented	Set when all eight bits of the ALU result are zero
5	K / X5	Underflow	Undocumented	Carry out of the 16-bit incrementer/decrementer, or signed comparison result
4	AC	Auxiliary carry	Documented	Carry out of bit 3 of the 8-bit ALU result; used by DAA
3	—	Unused	Reads as 0	Driven low when the flag register is read
2	P	Parity	Documented	Set for an even number of ones in the 8-bit ALU result

Bit	Symbol	Name	Status	Meaning and source
1	V	Overflow	Undocumented	Signed two's-complement overflow; reads as 1 on the 8080, hence 02H for a cleared flag byte in many simulators
0	CY	Carry	Documented	Carry or borrow out of the most significant bit of the operation

The Program Status Word is the 16-bit quantity formed by concatenating the accumulator and the flag register, and it is the entity manipulated by PUSH PSW and POP PSW:

$$PSW = (A \times 2^8) + F \quad (1)$$

where A is the accumulator and F the flag byte. The accumulator occupies the high-order byte and the flags the low-order byte. An instruction therefore changes the PSW if it changes the accumulator, if it changes any flag, or both. This definition is used in Section VII to establish exactly how much of the PSW a DAD instruction disturbs.

A note on the undocumented flags is warranted because learners meeting simulator output for the first time are often puzzled by it. On the original 8080, bit 1 of the flag byte always reads as one; many 8085 simulators reproduce this convention, so a flag register with every documented flag reset is displayed as 02H rather than 00H. On genuine 8085 silicon bit 1 carries the V flag. Neither V nor K is part of the documented programming model, and no portable program should depend on them; they are mentioned here only so that observed simulator values can be interpreted correctly.

IV. THE ACCUMULATOR-CENTRIC 8-BIT ARITHMETIC PATH

A. Structure of an 8-Bit ALU Operation

The arithmetic and logical instructions that operate on data share a single structure. One operand is implicitly the accumulator; the second is a register, the memory byte addressed by HL, or an immediate byte carried in the instruction. The operands are routed into ACT and TMP, the 8-bit ALU is driven for one pass, the result is written back to the accumulator, and all five documented flags are updated from the result and from the carry chain:

$$A \leftarrow A \oplus operand, \quad \{S, Z, AC, P, CY\} \leftarrow f(result) \quad (2)$$

where $\oplus$ denotes the selected operation. Instructions of this class—ADD r, ADC r, ADI data, SUB r, SBB r, ANA r, ORA r, XRA r and CMP r—execute in four T-states when the second operand is a register, and in seven when it is a memory byte or immediate data, because an additional read machine cycle is required [1], [3].

It is this uniform structure that gives rise to the rule taught in every introductory course: the accumulator is mandatory for ALU-based data arithmetic. The rule is correct within its scope. The error made by learners is not the rule itself but its unbounded generalisation to every instruction that performs an addition.

B. Multi-Byte Addition on an 8-Bit Path

Because the data path is eight bits wide, an addition of two 16-bit operands held in register pairs must be decomposed into a low-byte addition followed by a high-byte addition that consumes the carry produced by the first. The ADC instruction exists for exactly this purpose. Fig. 1 shows the generic control flow of such an addition, and Fig. 2 specialises it to a 16-bit operand pair held in memory.

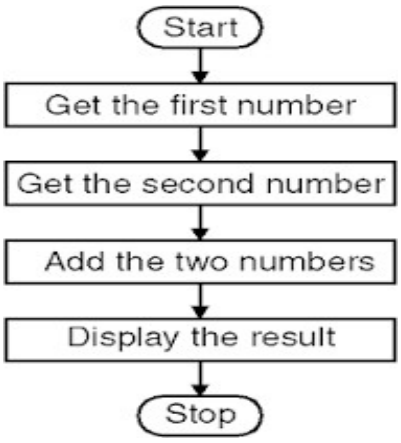

Fig. 1. Generic control flow of an addition routine: acquire the first operand, acquire the second, perform the addition and deliver the result.

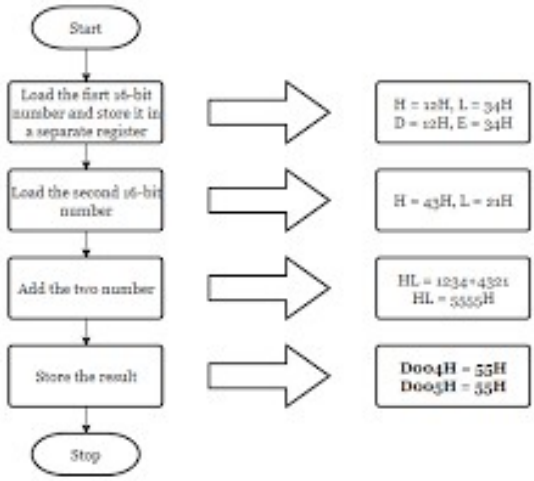

Fig. 2. The same flow specialised to 16-bit operands held in register pairs, illustrated with 1234H + 4321H = 5555H stored low byte first.

Expressed in instructions, the accumulator-based form of HL ← HL + DE is the six-instruction sequence MOV A,L; ADD E; MOV L,A; MOV A,H; ADC D; MOV H,A. Each step is individually simple, but the sequence has three properties that matter for address arithmetic: it occupies six bytes of program memory, it costs 24 T-states, and it destroys the previous contents of the accumulator while overwriting all five condition flags. The last of these is the most troublesome in practice, because address computations are frequently interleaved with data processing whose partial results and flag state must survive.

C. Why This Is Inadequate for Address Arithmetic

Address manipulation is not an occasional activity in an 8085 program; it is continuous. Traversing an array increments a pointer on every iteration, a stack frame is built and dismantled on every call, a jump table is indexed by scaling and adding a displacement, and a base address is combined with an offset whenever a data structure is accessed. If every such computation had to pass through the accumulator, the accumulator would have to be saved and restored around each one, and the flag state—on which the surrounding control flow depends—would be destroyed by work that is merely bookkeeping.

The designers of the 8080 and 8085 therefore provided a small group of 16-bit operations, of which DAD is the general one, that carry out address arithmetic without disturbing the data-processing state. Understanding this design intent is the key that makes the flag behaviour of DAD appear reasonable rather than arbitrary: DAD does not update S, Z, AC and P because updating them would defeat the purpose for which the instruction exists.

V. THE DAD INSTRUCTION: SEMANTICS, ENCODING AND TIMING

A. Semantics

The mnemonic DAD abbreviates “Double ADd”, where “double” refers to the double-width (16-bit) operand rather than to any doubling of the operand value. The instruction adds the contents of a specified register pair to the HL pair and leaves the sum in HL:

$$HL \leftarrow HL + rp, \quad rp \in \{BC, DE, HL, SP\} \tag{3}$$

DAD H therefore doubles the contents of HL, which is the standard idiom for a left shift of a 16-bit quantity, and DAD SP adds the stack pointer to HL, which is used to form addresses of items within the current stack frame. The accumulator is neither a source nor a destination. Of the documented flags, only Carry is affected; it is set when the 16-bit addition produces a carry out of bit 15 and reset otherwise [1], [2].

B. Encoding and Execution Profile

DAD occupies one byte. The register pair is encoded in bits 5 and 4 of the opcode, giving the four opcodes shown in Table II. Every form of the instruction executes in three machine cycles and ten T-states: an opcode fetch of four T-states followed by two bus-idle machine cycles of three T-states each, during which no external bus activity takes place because the work is entirely internal.

TABLE II. Encoding and execution profile of the DAD instruction

Mnemonic	Opcode	Operation	Bytes	Machine cycles	T-states
DAD B	09H	HL ← HL + BC	1	3 (fetch + 2 idle)	10 (4 + 3 + 3)
DAD D	19H	HL ← HL + DE	1	3 (fetch + 2 idle)	10 (4 + 3 + 3)
DAD H	29H	HL ← HL + HL	1	3 (fetch + 2 idle)	10 (4 + 3 + 3)
DAD SP	39H	HL ← HL + SP	1	3 (fetch + 2 idle)	10 (4 + 3 + 3)

These figures are fixed by the device and are independent of the clock frequency at which it is operated [21]. At the standard 8085A operating point of a 6.144 MHz crystal, the internal clock is 3.072 MHz and one T-state lasts 0.3255 µs, so a DAD instruction completes in approximately 3.26 µs. The same figures are used throughout Section VIII for cost comparison.

C. The Timing Anomaly That Demands Explanation

An ADD r instruction performs one 8-bit addition and needs four T-states, all of which are consumed by the opcode fetch; the arithmetic itself is completed within the fetch cycle. A DAD instruction fetches an opcode of the same length but then requires six additional T-states in which the external bus is idle. Two observations follow immediately. First, whatever DAD does, it does not need memory: no operand is read and no result is written outside the chip. Second, it needs substantially more internal time than a single 8-bit addition.

If the 8085 contained a 16-bit adder capable of producing the sum in one pass, there would be no reason for those six extra T-states; the operation would complete within the fetch cycle exactly as ADD does. The six idle T-states are therefore direct, externally observable evidence about the internal mechanism, and Section VI develops that evidence into the corrected model.

VI. RESOLVING THE CENTRAL MISCONCEPTION

A. The Question as Students Pose It

The question that motivates this paper is usually asked in the following form: if every ALU operation requires the accumulator, how can DAD add two 16-bit quantities without it? The question is well formed and deserves a precise answer rather than a reassurance.

B. The Common but Inaccurate Answer

The answer most often given is that the 8085 contains a separate internal 16-bit adder, sometimes called an address adder, which DAD uses instead of the main ALU, and which also serves INX, DCX, stack pointer updates and program counter increments. This account is attractive because it disposes of the contradiction in one sentence, but it does not survive contact with either the silicon or the timing data.

Reverse engineering of the 8085 die shows a single arithmetic unit, eight bits wide, built from eight one-bit slices [6], [7]. The only 16-bit arithmetic circuit on the chip is the increment/decrement network beneath the register file, and it is a chain of half-adders that can add or subtract a small constant, not an adder that accepts an arbitrary 16-bit operand [9]. Since DAD must add an arbitrary register pair, it cannot be served by that circuit. Furthermore, if a 16-bit adder existed, DAD would not require six bus-idle T-states beyond its fetch, and INX—which genuinely does use the 16-bit incrementer—would not be faster than DAD, yet INX completes in six T-states against ten for DAD.

C. The Accurate Answer

The correct explanation has two parts, and both are necessary.

1) The accumulator is bypassed, not the ALU:

The ALU never reads the accumulator directly. It reads the hidden ACT and TMP registers, and ACT is normally loaded from the accumulator. For DAD, and for the undocumented instructions of the same family such as DSUB and ARHL, the control logic loads ACT from a register-pair byte instead. The arithmetic is performed by the ordinary 8-bit ALU; what is bypassed is the accumulator, not the arithmetic unit [7]. The rule that “ALU operations use the accumulator” is thus a statement about the data-routing conventions of the documented data instructions, not a physical constraint of the ALU.

2) The 16-bit addition is performed as two 8-bit passes:

Because the ALU is eight bits wide, the 16-bit sum is produced in two sequential passes, exactly as a programmer would do it with ADD and ADC, but under microcode control and without touching the accumulator or the documented flags:

$$L \leftarrow L + rp_l, \quad c \leftarrow carry_8 \tag{4}$$

$$H \leftarrow H + rp_h + c, \quad CY \leftarrow carry_{16} \tag{5}$$

The intermediate carry c is held internally and is not exposed in the flag register; only the final carry out of bit 15 reaches CY. The three machine cycles of the instruction map onto this description naturally: the opcode fetch, then one bus-idle cycle for the low-order pass and one for the high-order pass. This is why DAD costs 10 T-states and not 4, and why those extra T-states are idle on the external bus.

D. The Three Arithmetic Resources Distinguished

The conflation criticised in Section VI-B disappears once three distinct resources are named separately. Table III sets them out.

TABLE III. The three arithmetic resources of the 8085 and the instructions each serves

Resource	Width	Operand source	Flags affected	Instructions served
8-bit ALU, accumulator path	8 bits	ACT loaded from A; TMP from register, memory or immediate data	S, Z, AC, P, CY	ADD, ADC, ADI, ACI, SUB, SBB, ANA, ORA, XRA, CMP, INR, DCR, DAA, RLC, RRC
8-bit ALU, accumulator-bypass path	8 bits, used twice	ACT loaded from a register-pair byte, not from A	CY only	DAD (and the undocumented DSUB, ARHL, LDHI, LDSI, RDEL)
16-bit incrementer/decrementer at the address latch	16 bits	Address latch; adds or subtracts a fixed constant only	None documented (drives the undocumented K flag)	INX, DCX, PC increment, SP adjustment during PUSH, POP, CALL, RET

The two data paths of interest can be written compactly. For 8-bit data arithmetic:

```
Register / Memory / Immediate  →  TMP
Accumulator                    →  ACT
ACT, TMP  →  8-bit ALU  →  Accumulator, all five flags
```

For the 16-bit address arithmetic of DAD:

```
Pass 1:  L, rp_low   → ACT, TMP → 8-bit ALU → L, internal carry
Pass 2:  H, rp_high, internal carry → ACT, TMP → 8-bit ALU → H, CY
```

Neither pass reads or writes the accumulator, and neither pass is permitted to update S, Z, AC or P. The single arithmetic unit is shared by both paths; only the routing and the flag-update permissions differ.

E. Why the Design Choice Is Rational

Two questions naturally follow: why not make the ALU 16 bits wide, and why suppress four of the five flags? The first is answered by the economics of 1976 NMOS fabrication. Doubling the width of the ALU and of the associated temporary registers and control lines would have consumed die area and power for a benefit confined to a small number of instructions, while the two-pass approach reuses hardware that already exists at a cost of six T-states per instruction. The second is answered by the intended use of the instruction. DAD exists to compute addresses in the middle of data processing; if it wrote S, Z, AC and P, then every pointer update would destroy the condition under test, and the programmer would have to save and restore the flag state around it—precisely the overhead the instruction was introduced to remove. Suppressing the flag update is not an oversight but a deliberate feature. Carry alone is retained because a carry out of bit 15 signals an address wrap, which the program may legitimately need to detect.

VII. FLAG AND PSW BEHAVIOUR DURING 16-BIT ADDITION

A. Derivation of the Flag Effects

The flag effects of DAD follow from the model of Section VI rather than having to be memorised. The Sign, Zero, Auxiliary Carry and Parity flags are defined over an 8-bit ALU result associated with the accumulator. During a DAD instruction two 8-bit results are produced, neither of which is the accumulator and neither of which is the whole 16-bit sum; there is therefore no well-defined 8-bit quantity for these four flags to describe, and the control logic does not update them. The Carry flag, by contrast, has an unambiguous meaning for a 16-bit addition—the carry out of bit 15—and it is updated.

It is worth stating explicitly what the Zero flag would mean if it were updated, because this is the single most common error. If Z tracked the low-order pass, it would be set whenever the low byte of the sum happened to

be zero, which is almost never what the programmer wants. If it tracked the high-order pass, it would be set whenever the high byte was zero. Neither corresponds to "the 16-bit result is zero". A flag that reported either of these would be actively misleading, and the decision not to update Z at all is the safer engineering choice.

TABLE IV. Flag effects of representative 8085 instructions

Instruction	S	Z	AC	P	CY	Remark
`ADD r`	Y	Y	Y	Y	Y	Accumulator is source and destination
`ADC r`	Y	Y	Y	Y	Y	Adds the incoming CY as a third operand
`SUB r`	Y	Y	Y	Y	Y	CY acts as a borrow indicator
`INR r`	Y	Y	Y	Y	N	CY deliberately preserved for multi-byte loops
`DCR r`	Y	Y	Y	Y	N	CY deliberately preserved
`INX rp`	N	N	N	N	N	Uses the 16-bit incrementer; no documented flag changes
`DCX rp`	N	N	N	N	N	A zero result cannot be detected with JZ
`DAD rp`	N	N	N	N	Y	Only the carry out of bit 15 is recorded
`CMP r`	Y	Y	Y	Y	Y	Flags only; the accumulator is unchanged

Y indicates that the flag is updated from the result of the operation; N indicates that the flag retains its previous value.

B. Consequences for the PSW

Applying (1), a DAD instruction leaves the high-order byte of the PSW—the accumulator—completely unchanged, and modifies at most bit 0 of the low-order byte. Fifteen of the sixteen PSW bits are therefore guaranteed to be preserved, and the sixteenth changes only if the addition carries out of bit 15. In the language of the programming model, PSW is partially affected, and the part affected is precisely the Carry flag.

This has three practical consequences that are worth drawing out for students:

- A PUSH PSW executed after a DAD saves the same accumulator value that was current before the DAD, and a flag byte that differs at most in bit 0. Sequences that save the PSW before a block of address arithmetic and restore it afterwards are therefore usually unnecessary, though they remain necessary if the Carry flag itself must survive.
- A conditional branch on Z, S or P placed immediately after a DAD does not test the DAD at all; it tests whatever operation last updated those flags, which may be many instructions earlier. This produces programs that appear to work on one data set and fail on another, and it is a common source of laboratory debugging time.
- Only JC and JNC are meaningful immediately after a DAD, and they test the 16-bit carry out. To branch on a zero 16-bit result the programmer must construct the test explicitly, for example MOV A,H; ORA L; JZ target, which costs three further instructions and does update the flags from the accumulator.

C. A Note on DAD H and Shifting

Since DAD H computes HL + HL, it is the idiomatic 16-bit left shift, and the Carry flag receives the bit shifted out of position 15. Repeated application provides multiplication by powers of two, which is the standard technique for scaling an array index by the element size before adding it to a base address. This idiom is a useful vehicle for teaching because it exercises the carry semantics of DAD in a context where the carry genuinely matters.

VIII. WORKED EXAMPLES AND QUANTITATIVE COMPARISON

A. Traced Cases

Table V traces six representative cases through the two-pass model of (4) and (5). In each case the two 8-bit passes are shown so that the origin of the final Carry value is visible, and the state of the remaining flags is reported explicitly as unchanged.

TABLE V. Traced DAD operations showing both 8-bit passes

Case	HL before	Operand	Pass 1 (low byte)	Pass 2 (high byte)	CY	Observation
1	1234H	DE = 1111H	34+11 = 45, c=0	12+11+0 = 23	0	Straightforward sum 2345H; S, Z, AC and P retain their earlier values
2	1234H	DE = 1312H	34+12 = 46, c=0	12+13+0 = 25	0	Sum 2546H; no carry out of bit 15
3	12FFH	DE = 0001H	FF+01 = 00, c=1	12+00+1 = 13	0	Internal carry crosses the byte boundary but never reaches CY or AC
4	FFFFH	DE = 0001H	FF+01 = 00, c=1	FF+00+1 = 00	1	Result 0000H with CY set; the Zero flag is NOT set
5	8000H	HL (DAD H)	00+00 = 00, c=0	80+80+0 = 00	1	Left shift of HL; the bit shifted out of position 15 appears in CY
6	0012H	DE = 0034H	12+34 = 46, c=0	00+00+0 = 00	0	The case verified on the simulator in Section IX; result 0046H

Case 4 is the decisive one for teaching. The 16-bit result is zero, and a learner who believes that DAD updates the flags in the ordinary way will predict Z = 1 and will write a JZ instruction to detect the condition. The prediction is wrong, and the program will branch or fail to branch according to the state left by some earlier instruction. Case 3 is the second most useful, because it shows an internal carry propagating from bit 7 into bit 8 without appearing anywhere in the flag register; it also makes clear that AC, which reports a carry out of bit 3 of an 8-bit accumulator operation, has no meaning here and is left alone.

B. Cost of the Two Formulations

The value of DAD can be quantified. Table VI compares the arithmetic core of a 16-bit addition, HL ← HL + DE, expressed first with DAD and then with the accumulator-based sequence of Section IV-B, together with the complete memory-to-memory programs of Section IX. Timing is given at the 3.072 MHz internal clock of a standard 8085A.

TABLE VI. Cost comparison of the two formulations of a 16-bit addition

Metric	DAD form	ADD/ADC form	Difference	Remark
Instructions in the arithmetic core	1	6	6× fewer	DAD D against MOV/ADD/MOV/MOV/ADC/MOV
Bytes in the arithmetic core	1	6	5 bytes saved	Significant in a 2 KB monitor ROM
T-states in the arithmetic core	10	24	2.4× faster	3.26 µs against 7.81 µs
Bytes, complete program	12	17	29.4% smaller	Listings 1 and 2
T-states, complete program	67	81	17.3% faster	21.81 µs against 26.37 µs
Accumulator after execution	Preserved	Destroyed	—	Decisive when arithmetic is interleaved with data processing
S, Z, AC, P after execution	Preserved	Overwritten	—	Surrounding conditional logic survives a DAD
Carry semantics	Carry out of bit 15	Carry out of bit 15	Equivalent	Both report 16-bit overflow correctly

The time and space savings are worthwhile but not dramatic; the preservation of the accumulator and of four condition flags is the property that changes how programs are written. A pointer can be advanced in the middle of a comparison loop without saving anything,

which is exactly the pattern that address arithmetic demands.

IX. PROGRAM LISTINGS AND SIMULATOR VERIFICATION

A. Listing 1: 16-Bit Addition Using DAD

The program reads a 16-bit operand from 2050H–2051H, a second from 2052H–2053H, adds them and stores the 16-bit sum at 2060H–2061H. All 16-bit quantities are held in memory in the little-endian order used by the 8085, low byte first. The program is assembled at 0800H.

LISTING 1. 16-bit addition using the DAD instruction

Address	Machine code	Mnemonic	T-states	Comment
0800H	2A 50 20	LHLD 2050H	16	HL ← first operand from 2050H–2051H
0803H	EB	XCHG	4	DE ← first operand, freeing HL
0804H	2A 52 20	LHLD 2052H	16	HL ← second operand from 2052H–2053H
0807H	19	DAD D	10	HL ← HL + DE; only CY is updated
0808H	22 60 20	SHLD 2060H	16	Store the 16-bit sum at 2060H–2061H
080BH	76	HLT	5	Halt; PC stops at 080CH

Total: 12 bytes, 67 T-states, 21.81 µs at 3.072 MHz.

The XCHG instruction deserves comment because its necessity is not obvious. LHLD can only load HL, so the first operand must be moved out of HL before the second is loaded. XCHG exchanges HL with DE in four T-states without using memory or the accumulator, which makes it the natural choice; the alternative, MOV D,H followed by MOV E,L, costs eight T-states and two bytes instead of four T-states and one byte.

If a carry out of bit 15 must be recorded, three further instructions suffice and they do not disturb the stored sum:

```
MVI A, 00H      ; A ← 0
ACI 00H         ; A ← 0 + 0 + CY, so A
becomes 0000000CY
STA 2062H       ; store the carry byte
```

Here the accumulator is used deliberately, and the ACI instruction consumes the Carry flag left by DAD. This is a good demonstration that the two arithmetic paths cooperate rather than compete: DAD produces the carry, and an ordinary accumulator instruction captures it.

B. Listing 2: The Same Computation Without DAD

For comparison, the same computation written entirely on the accumulator path is given in Listing 2. It produces an identical sum at 2060H and an identical Carry flag, but it destroys the accumulator and all five flags along the way.

LISTING 2. The same 16-bit addition using only accumulator arithmetic

Address	Machine code	Mnemonic	T-states	Comment
0800H	2A 50 20	LHLD 2050H	16	HL ← first operand
0803H	EB	XCHG	4	DE ← first operand
0804H	2A 52 20	LHLD 2052H	16	HL ← second operand
0807H	7D	MOV A, L	4	A ← low byte of the second operand
0808H	83	ADD E	4	Low-order addition; CY holds the inter-byte carry
0809H	6F	MOV L, A	4	Store the low byte of the sum
080AH	7C	MOV A, H	4	A ← high byte of the second operand
080BH	8A	ADC D	4	High-order addition with the carry included
080CH	67	MOV H, A	4	Store the high byte of the sum
080DH	22 60 20	SHLD 2060H	16	Store the 16-bit sum
0810H	76	HLT	5	Halt

Total: 17 bytes, 81 T-states, 26.37 µs at 3.072 MHz.

Placing the two listings side by side is pedagogically effective because the six instructions from 0807H to 080CH in Listing 2 are a visible, executable expansion of the single DAD D instruction at 0807H in Listing 1 — which is, in outline, what the microcode of the processor does internally during the two bus-idle machine cycles. The one difference is that the internal version is not permitted to touch the accumulator or the four suppressed flags.

C. Simulator Verification

Listing 1 was executed on a browser-based 8085 simulator with the program loaded at 0800H and the operands placed at 2050H and 2052H. The operands used for the run were 0034H at 2050H and 0012H at 2052H, corresponding to Case 6 of Table V. Fig. 3 shows the machine state after the HLT instruction.

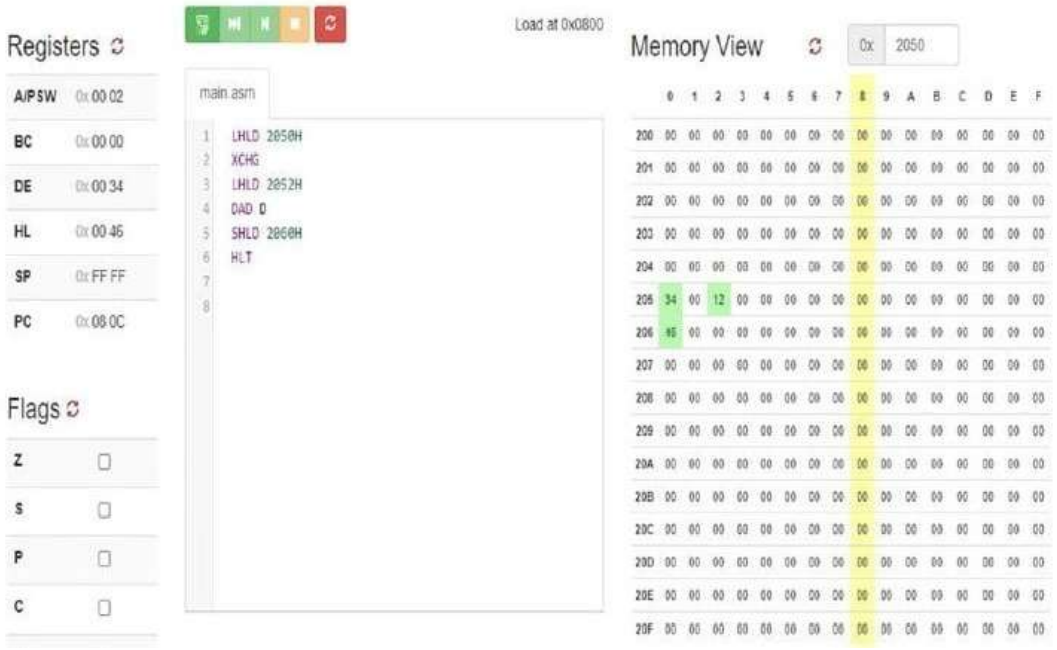

Fig. 3. State of the simulated 8085 after executing Listing 1. HL holds the sum 0046H, DE retains the first operand 0034H, the accumulator is unchanged at 00H, all documented flags are reset, and the result byte 46H is visible at 2060H in the memory view.

Four features of the displayed state confirm the analysis of Sections VI and VII:

1) HL contains 0046H, the sum of 0034H and 0012H, and the byte 46H appears at address 2060H in the memory view, confirming that SHLD wrote the result correctly.
2) DE still contains 0034H. DAD reads its source register pair but does not modify it.
3) The accumulator is 00H, its value before the program ran. No instruction in Listing 1 writes the accumulator, which confirms directly that the 16-bit addition took place without it.
4) The Z, S, P and C checkboxes are all clear, and the combined A/PSW display reads 0002H. With A = 00H the flag byte is 02H rather than 00H because the simulator reproduces the 8080 convention in which bit 1 of the flag register always reads as one, as noted in Section III-E. The Carry flag is correctly reset, since 0034H + 0012H does not overflow 16 bits.

The program counter halts at 080CH, one byte past the HLT at 080BH, which is consistent with the 12-byte program length reported in Listing 1. Repeating the run with the operands of Case 4 in Table V, FFFFH and 0001H, produces HL = 0000H with the C indicator set and the Z indicator still clear, which is the observation students find most surprising and which is the most effective single demonstration of the flag semantics of DAD.

X. MISCONCEPTIONS, DIAGNOSES AND REMEDIES

Table VII collects the misconceptions encountered most often in connection with this material. Each entry pairs the incorrect belief with the correct statement and with a short diagnostic exercise that exposes the error, following the standard practice of confronting a faulty mental model with the case it cannot explain [4], [5].

TABLE VII. Common misconceptions and the exercises that expose them

Misconception	Correct position	Diagnostic exercise
Every arithmetic instruction must use the accumulator.	Only the documented data-arithmetic instructions do. DAD reaches the same ALU through the ACT register without involving A.	Load A with AAH, execute DAD D, then display A. It is still AAH.
DAD uses a separate 16-bit adder.	The 8085 has one 8-bit ALU. DAD uses it twice, which is why it needs six bus-idle T-states beyond its fetch.	Compare the timing of ADD r (4 T), DAD rp (10 T) and INX rp (6 T) and account for the difference.
DAD, INX and PC stepping all use the same 16-bit unit.	INX, DCX and PC stepping use the 16-bit incrementer at the address latch; DAD cannot, because that circuit adds only a fixed constant.	Explain why INX takes 6 T-states while DAD takes 10, if both used the same circuit.
The Zero flag is set when HL becomes 0000H after DAD.	Z is defined over an 8-bit accumulator result and is not updated by DAD at all.	Execute Case 4 of Table V and observe Z remaining clear while HL is 0000H.
JZ can be used to detect a zero 16-bit sum.	The test must be constructed: MOV A,H; ORA L; JZ target.	Write both versions and run them with HL = 0000H after a DAD.
DAD changes the PSW completely.	DAD preserves the accumulator and four flags, so at most one of the sixteen PSW bits changes.	PUSH PSW before and after a DAD and compare the two stacked words.
INX affects the flags in the same way as INR.	INR updates S, Z, AC and P; INX updates no documented flag.	Execute INX H with HL = FFFFH and observe that Z and CY are both unchanged.
DAD H doubles the value in H only.	DAD H adds the whole HL pair to itself, giving a 16-bit left shift with the bit lost from position 15 appearing in CY.	Execute DAD H with HL = 8000H and inspect HL and CY.

XI. EDUCATIONAL DEPLOYMENT

A. Suggested Sequence of Instruction

The material has been arranged so that the correct model is reached by extending a structure the learner already holds, in line with the load-reduction principle of [18]. A single ninety-minute session covers it comfortably in five stages. The session begins with the familiar accumulator path and the rule that follows from it. It then introduces the 16-bit addition problem and asks the class to write it with ADD and ADC, producing Listing 2 and, with it, first-hand experience of the accumulator being destroyed. DAD is then introduced as the instruction that solves this problem, and the class is invited to predict its flag behaviour, at which point the misconception surfaces of its own accord. The prediction is tested on the simulator using Case 4, and the resulting contradiction motivates the corrected model of Section VI, which is presented last, when the learner has a reason to want it. The session closes with the timing comparison of Table VI, which converts the qualitative argument into numbers.

B. Learning Outcomes and Assessment

Table VIII states the intended learning outcomes with the cognitive level at which each is assessed and a matching assessment item. The items are written so that they cannot be answered by recall of a single sentence from a textbook.

TABLE VIII. Learning outcomes and matched assessment items

No.	On completion, the learner can	Cognitive level	Assessment item
1	Distinguish data arithmetic from address arithmetic and name the instructions belonging to each.	Understand	Classify ADD B, DAD B, INX H, INR L and ADC M by the arithmetic resource each uses, and justify each classification.
2	Explain how DAD performs a 16-bit addition without the accumulator.	Understand	Explain why DAD requires ten T-states while ADD requires four, and what the six additional T-states are spent on.
3	Predict the flag and PSW state after any 16-bit arithmetic instruction.	Apply	Given HL = FFF0H, DE = 0020H and CY = 0, state HL, CY, Z and A after DAD D, with reasons.
4	Write correct code to detect a zero or overflowing 16-bit result.	Apply	Extend Listing 1 so that it stores 01H at 2062H if the sum is zero and 00H otherwise.
5	Compare implementation alternatives quantitatively.	Analyse	Compute the byte count and T-state count of both forms of a 16-bit addition and state the conditions under which the shorter form is not preferable.
6	Detect and repair flag-related defects in supplied code.	Evaluate	A supplied routine uses JZ immediately after DAD D to terminate a loop. Explain why it fails intermittently and correct it.

C. Transfer to Later Topics

The distinction drawn here is not confined to the 8085. The separation of a narrow data path from wider address arithmetic reappears as the address generation unit of the 8086 and as the dedicated address arithmetic logic of modern pipelined processors, where address computation is deliberately removed from the main integer unit for the same reason the 8085 removes it from the accumulator: to avoid serialising bookkeeping behind data processing. Likewise, the suppression of flag updates by DAD is an early instance of the general principle that an architecture should not couple unrelated state, a principle visible today in instruction sets that provide both flag-setting and non-flag-setting forms of the same arithmetic operation. Presenting the 8085 case carefully therefore pays a return when these later topics are introduced.

D. Relevance to Contemporary Resource-Constrained Computing

The habits of reasoning that this material builds — counting bytes, counting cycles and asking what machine state an operation destroys — do not expire with the device that introduced them. They are the everyday arithmetic of embedded and edge deployment, where an algorithm must fit inside a fixed memory footprint and a fixed energy budget on a node that has other work to do as well. Trust-management schemes for mobile ad hoc and Internet-of-Things networks, for example, evaluate a trust metric on every routing decision, and they execute on nodes whose computational capability is far closer to that of an 8085 than to that of a workstation [22]. On-device perception systems face the same pressure: an assistive device that recognises objects and synthesises speech in real time [23], or a pipeline that detects activity and counts people in a live video stream [24], succeeds or fails on precisely the quantities tabulated in Table VI — instruction count, cycle count, and the machine state that must be preserved across each operation.

The same accounting explains design choices in lightweight security. Elliptic curve cryptography is preferred over integer-factorisation schemes on constrained ad hoc and IoT nodes because it delivers comparable security with much smaller keys and therefore far fewer arithmetic operations per exchange, which is why it recurs in protocols intended for such networks [25], [26]. A learner who has just counted twenty-four T-states for a single software 16-bit addition on an 8-bit data path has a concrete basis for the claim that key length translates into energy drawn from a battery; a learner who has been told only that a scheme is efficient does not.

Machine-learning workloads make the point most sharply, because inference is increasingly pushed toward the edge rather than kept in the data centre. Intrusion detection built on dimensionality reduction and ensemble classifiers [27], traffic prediction with recurrent models [28], and transfer-learned convolutional detectors applied to objects, clinical images and other visual targets [29]–[32] are feasible or infeasible on a given platform according to the number of multiply–accumulate operations required and the width of the arithmetic the hardware provides natively. The two-pass execution of DAD analysed in Section VI is the same phenomenon in miniature as an 8-bit microcontroller emulating 32-bit floating-point arithmetic in software: when the data path is narrower than the operand, the operation is decomposed and the cost multiplies. A student who has traced that mechanism once at a scale small enough to follow by hand recognises it later at a scale that cannot be traced at all.

Finally, the pedagogical move made in Section X — confronting a faulty model with the case it cannot explain — generalises well beyond processor architecture. A misconception about what an abstraction does internally is a recurring source of defects at every layer of a system: assumptions about how a document-store query interface treats the input it is handed produce injection surfaces [33], and assumptions about what a distributed storage platform protects by default produce exposure in large-scale deployments [34]. In each case the remedy has the same shape as the one adopted here, namely to make the internal mechanism visible at the precise point where the learner's existing model first fails to predict the observed behaviour.

E. A Note on Laboratory Delivery

Where the laboratory is delivered through browser-based simulators hosted centrally rather than through individual hardware trainer kits, a practical question arises alongside the pedagogical one: how many concurrent student sessions the hosting infrastructure can sustain during a timetabled slot. The answer is governed by the allocation policy applied to the underlying pool, and the dynamic and threshold-based allocation schemes studied for cloud environments [35], [36] apply directly, because laboratory load is bursty and strongly correlated with the timetable rather than uniformly distributed. Institutions adopting the sequence of Section XI-A for large cohorts should therefore provision against peak concurrent sessions rather than against enrolment, and should retain a hardware trainer path for students whose connectivity is unreliable. This is an operational rather than a

conceptual matter, but it determines whether the simulator step of Section XI-A — the step on which the confrontation with the misconception depends — actually takes place.

XII. LIMITATIONS AND FUTURE WORK

Three limitations should be stated plainly. First, the account of the internal mechanism rests on published instruction timing together with die-level reverse engineering reported by others [6]–[10]; the present authors have not independently verified the silicon, and Intel never documented the microcode. The two-pass model is strongly supported by the ten-T-state, six-idle execution profile and by the width of the ALU, but it remains an inference about an undocumented implementation rather than a manufacturer statement. Nothing in the documented programming model depends on the inference: the flag and PSW behaviour described in Section VII is specified by Intel and holds regardless of how the sum is produced internally.

Second, this is a conceptual and analytical paper rather than an empirical study of learning. The misconceptions in Table VII are drawn from classroom and laboratory experience and are consistent with what the misconception literature reports in adjacent subjects, but no controlled measurement of the effect of this presentation on student performance has been carried out. A pre-test and post-test study using the assessment items of Table VIII, administered across parallel sections with and without the corrected model, would establish whether the explanation improves outcomes and by how much.

Third, the treatment is confined to addition. DAD has no subtracting counterpart in the documented instruction set; 16-bit subtraction must be synthesised from the accumulator path or, on genuine 8085 silicon, from the undocumented DSUB instruction, which is not portable to the 8080 and is not available on all simulators. A parallel analysis of 16-bit subtraction, comparison and shifting would complete the picture.

Beyond addressing these limitations, three extensions are planned: an interactive visualisation that animates the two ALU passes of a DAD instruction alongside a live flag display, so that the internal carry can be watched crossing the byte boundary; a short diagnostic concept inventory for 8085 arithmetic and flag semantics, validated on multiple cohorts; and an extension of the same treatment to the 8086, where the contrast between the ALU and the address generation unit can be developed further.

XIII. CONCLUSION

The apparent contradiction between the accumulator-centric arithmetic of the Intel 8085 and the existence of an accumulator-free 16-bit addition is resolved once three arithmetic resources are distinguished rather than conflated. The accumulator path, in which ACT is loaded from the accumulator and all five documented flags are updated, serves data arithmetic. The accumulator-bypass path, in which ACT is loaded from a register-pair byte and the same 8-bit ALU is driven twice with an internal carry between the passes, serves DAD. The 16-bit incrementer at the address latch, which can add or subtract only a fixed constant, serves INX, DCX and program counter stepping. The widely repeated claim that DAD uses a dedicated 16-bit adder belongs to none of these and is contradicted both by the ten-T-state execution profile of the instruction and by die-level analysis of the chip.

From this model the flag behaviour follows as a consequence rather than as a rule to be memorised. Sign, Zero, Auxiliary Carry and Parity are defined over an 8-bit accumulator result that a DAD instruction never produces, so they are not updated; Carry has an unambiguous 16-bit meaning, so it is. The Program Status Word is therefore preserved except, at most, in its Carry bit, and the practical consequences for PUSH PSW, for conditional branching and for zero-result detection follow directly. The quantitative comparison shows that the DAD form of a 16-bit addition costs one byte and ten T-states against six bytes and twenty-four T-states for the accumulator-based equivalent, while additionally preserving the accumulator and four condition flags — the last of which is the property that makes address arithmetic practical in the middle of data processing.

Presented in this order, with the simulator evidence of Section IX and the diagnostic exercises of Table VII, the material converts a persistent source of confusion into an opportunity to teach something of lasting value: that an instruction set is a negotiated interface to hardware whose structure it reflects but does not fully reveal, and that the behaviour of a flag is a design decision with reasons behind it.

FUNDING DECLARATION

The authors declare that no financial support, funding, grant or sponsorship was received from any public, private, governmental or non-governmental agency for the conduct of this work. The study was carried out independently using the institutional resources available to the authors. No external funding influenced the design of the study, the analysis, the

interpretation of results or the preparation of the manuscript.

CONFLICT OF INTEREST

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

DATA AND CODE AVAILABILITY

This work uses no external data set. The two assembly language listings presented in Section IX are reproduced in full within the paper and can be assembled and executed on any standard 8085 trainer kit or browser-based simulator. The simulator state shown in Fig. 3 is reproducible by loading Listing 1 at 0800H with 34H, 00H at 2050H–2051H and 12H, 00H at 2052H–2053H.

AUTHOR CONTRIBUTIONS

S. Chouhan carried out the architectural analysis, prepared the listings, performed the simulator verification and drafted the manuscript. G. K. Prajapati supervised the work, proposed the instruction-timing argument and revised the technical content. N. Khare developed the pedagogical framing, the misconception table and the assessment items, and edited the manuscript. All authors read and approved the final version.

REFERENCES

[1] R. S. Gaonkar, Microprocessor Architecture, Programming, and Applications with the 8085, 6th ed. New Delhi, India: Penram International Publishing, 2013.

[2] Intel Corporation, MCS-80/85 Family User's Manual. Santa Clara, CA, USA: Intel Corporation, 1979.

[3] Intel Corporation, 8080/8085 Assembly Language Programming Manual. Santa Clara, CA, USA: Intel Corporation, 1979.

[4] D. Hestenes, M. Wells, and G. Swackhamer, "Force concept inventory," The Physics Teacher, vol. 30, no. 3, pp. 141–158, Mar. 1992.

[5] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," Computer Science Education, vol. 13, no. 2, pp. 137–172, 2003.

[6] K. Shirriff, "Inside the ALU of the 8085 microprocessor," Ken Shirriff's Blog, Jan. 2013. [Online]. Available: http://www.righto.com/2013/01/inside-alu-of-8085-microprocessor.html

[7] K. Shirriff, "Reverse-engineering the 8085's ALU and its hidden registers," Ken Shirriff's Blog, Jul. 2013. [Online]. Available: http://www.righto.com/2013/07/reverse-engineering-8085s-alu-and-its.html

[8] K. Shirriff, "Silicon reverse engineering: The 8085's undocumented flags," Ken Shirriff's Blog, Feb. 2013. [Online]. Available: https://www.righto.com/2013/02/looking-at-silicon-to-understanding.html

[9] K. Shirriff, "The 8085's register file reverse engineered," Ken Shirriff's Blog, Mar. 2013. [Online]. Available: http://www.righto.com/2013/03/register-file-8085.html

[10] K. Shirriff, "Reverse-engineering the flag circuits in the 8085 processor," Ken Shirriff's Blog, Jul. 2013. [Online]. Available: http://www.righto.com/2013/07/reverse-engineering-flag-circuits-in.html

[11] T. R. Padmanabhan, Programming and Interfacing the 8085 Microprocessor. New Delhi, India: BPB Publications, 2007.

[12] W. Stallings, Computer Organization and Architecture: Designing for Performance, 10th ed. Boston, MA, USA: Pearson, 2016.

[13] J. P. Hayes, Computer Architecture and Organization, 3rd ed. New York, NY, USA: McGraw-Hill, 1998.

[14] A. K. Ray and K. M. Bhurchandi, Advanced Microprocessors and Peripherals, 3rd ed. New Delhi, India: McGraw-Hill Education, 2011.

[15] B. Ram, Fundamentals of Microprocessors and Microcomputers, 8th ed. New Delhi, India: Dhanpat Rai Publications, 2016.

[16] D. V. Hall, Microprocessors and Interfacing: Programming and Hardware, 2nd ed. New York, NY, USA: McGraw-Hill, 1992.

[17] L. Ma, J. Ferguson, M. Roper, and M. Wood, "Investigating and improving the models of programming concepts held by novice programmers," Computer Science Education, vol. 21, no. 1, pp. 57–80, 2011.

[18] J. Sweller, "Cognitive load during problem solving: Effects on learning," Cognitive Science, vol. 12, no. 2, pp. 257–285, 1988.

[19] M. M. Mano and M. D. Ciletti, Digital Design: With an Introduction to the Verilog HDL, 5th ed. Upper Saddle River, NJ, USA: Pearson, 2013.

[20] S. Salivahanan, Digital Circuits and Design. New Delhi, India: Oxford University Press, 2012.

[21] Intel Corporation, 8085AH/8085AH-2/8085AH-1 8-Bit HMOS Microprocessors Data Sheet. Santa Clara, CA, USA: Intel Corporation, 1990.

[22] U. Singh, M. Shukla, A. K. Jain, M. Patsariya, R. Itare, and S. Yadav, Trust Based Model for Mobile Ad-Hoc Network in Internet of Things, vol. 98. Singapore: Springer, 2020.

[23] P. Gupta, M. Shukla, N. Arya, U. Singh, and K. Mishra, "Let the blind see: An AIIoT-based device for real-time object recognition with the voice conversion," in Machine Learning for Critical Internet of Medical Things, F. Al-Turjman and A. Nayyar, Eds. Cham, Switzerland: Springer, 2022, doi: 10.1007/978-3-030-80928-7_8.

[24] U. Singh, P. Gupta, and M. Shukla, "Activity detection and counting people using Mask-RCNN with bidirectional ConvLSTM," Journal of Intelligent & Fuzzy Systems, vol. 42, no. 6, pp. 6505–6520, 2022.

[25] M. Shukla, B. K. Joshi, and U. Singh, "Mitigate wormhole attack and blackhole attack using elliptic curve cryptography in MANET," Wireless Personal Communications, vol. 121, pp. 503–526, 2021, doi: 10.1007/s11277-021-08647-1.

[26] M. Muwel, P. Mishra, M. Samvatsar, U. Singh, and R. Sharma, "Efficient ECGDH algorithm through protected multicast routing protocol in MANETs," in Proc. Int. Conf. Electronics, Communication and Aerospace Technology (ICECA), 2017, doi: 10.1109/ICECA.2017.8212743.

[27] S. Waskle, L. Parashar, and U. Singh, "Intrusion detection system using PCA with random forest approach," in Proc. Int. Conf. Electronics and Sustainable Communication Systems (ICESC), Coimbatore, India, 2020, pp. 803–808, doi: 10.1109/ICESC48915.2020.9155656.

[28] S. Nihale, S. Sharma, L. Parashar, and U. Singh, "Network traffic prediction using long short-term memory," in Proc. Int. Conf. Electronics and Sustainable Communication Systems (ICESC), Coimbatore, India, 2020, pp. 338–343, doi: 10.1109/ICESC48915.2020.9156045.

[29] B. Bamne, N. Shrivastava, L. Parashar, and U. Singh, "Transfer learning-based object detection by using convolutional neural networks," in Proc. Int. Conf. Electronics and Sustainable Communication Systems (ICESC), Coimbatore, India, 2020, pp. 328–332, doi: 10.1109/ICESC48915.2020.9156060.

[30] A. Saxena, A. Vyas, L. Parashar, and U. Singh, "A glaucoma detection using convolutional neural network," in Proc. Int. Conf. Electronics and Sustainable Communication Systems (ICESC), Coimbatore, India, 2020, pp. 815–820, doi: 10.1109/ICESC48915.2020.9155930.

[31] R. Baghel, P. Pahadiya, and U. Singh, "Human face mask identification using deep learning with OpenCV techniques," in Proc. 7th Int. Conf. Communication and Electronics Systems (ICCES), Coimbatore, India, 2022, pp. 1051–1057, doi: 10.1109/ICCES54183.2022.9835884.

[32] U. Singh and L. S. Songare, "Analysis and detection of monkeypox using the GoogLeNet model," in Proc. Int. Conf. Automation, Computing and Renewable Systems (ICACRS), Pudukkottai, India, 2022, pp. 1000–1008, doi: 10.1109/ICACRS55517.2022.10029125.

[33] U. Singh, N. K. Solanki, M. K. Varma, and T. Sevak, "Towards analyzing MongoDB NoSQL security and designing injection defense solution," in Proc. 2nd Int. Conf. Communication and Electronics Systems (ICCES), 2018, doi: 10.1109/CESYS.2017.8321222.

[34] U. Singh, N. K. Solanki, M. K. Varma, and T. Sevak, "A review on big data protection of Hadoop," in Proc. 2nd Int. Conf. Communication and Electronics Systems (ICCES), 2018, doi: 10.1109/CESYS.2017.8321221.

[35] P. Gupta, M. Samvatsar, and U. Singh, "Cloud computing through dynamic resource allocation scheme," in Proc. Int. Conf. Electronics, Communication and Aerospace Technology (ICECA), 2017, doi: 10.1109/ICECA.2017.8212723.

[36] P. S. Chouhan, M. Samvatsar, and U. Singh, "Energetic resource allotment scheme for cloud computing using threshold-based approach," in Proc. Int. Conf. Electronics, Communication and Aerospace Technology (ICECA), 2017, doi: 10.1109/ICECA.2017.8212744.

