FOREWORD

This research was conducted by the Electronic Systems Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, under Contract AF 33(616)-8363 with The 6570th Aerospace Medical Research Laboratories. This work was performed in support of Project No. 6114, "Training Equipment, Simulators, and Techniques for Air Force Systems," and Task No. 611408, "Simulation Computers." Work began on 1 May 1961 and will terminate on 30 April 1962. Mr. William Goeckler, Simulation Techniques Section, Training Research Branch of the Behavioral Sciences Laboratory, was the contract monitor.

This report is based on a thesis submitted by Louis Krasny in August 1961 in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering at the Massachusetts Institute of Technology, Cambridge, Massachusetts. It has also been published by the M. I. T. Electronic Systems Laboratory as Report ESL-R-118, Project DSR 8823.

The author wishes to express his appreciation to Professor Alfred K. Susskind for his patient supervision of this thesis and to Mark E. Connelly, project engineer, for his invaluable assistance in all phases of this study. Both are members of the Electronic Systems Laboratory at M. I. T.

ABSTRACT

To determine the ability of a moderate-sized digital computer, such as the M. I. T. TX-O, to solve a complex real-time flight simulation problem, the complete equations for the unrestricted simulation of the F-100A aircraft have been programmed using the TX-O order code. From an analysis of this program, specific recommendations are made for logical modifications of the TX-O to facilitate real-time simulation. With nine additional orders, including a 25μ sec. addressable multiply, a 40μ sec. divide, and a special level-sense order to facilitate nonlinear function generation, the TX-O would solve the full F-100A equations at 23 solutions per second.

The areas of function generation, order code specification, usage of subroutines, integration, word length requirement, input and output procedures, decision-making, and high-speed multiplication are investigated in detail with quantitative comparisons between different methods wherever possible.

Although the emphasis of this study has been on the specific requirements of the F-100A problem, the design methodology and the various design trade-offs described should be applicable to the functional analysis of other simulation problems of greater or lesser complexity.

PUBLICATION REVIEW

This technical documentary report has been reviewed and is approved.

Walter F. Grether

WALTER F. GRETHER
Technical Director
Behavioral Sciences Laboratory

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

LIST OF TABLES

SYMBOLS

A. AERODYNAMIC TERMS

X_s, Y_s, Z_s	aerodynamic forces along stability axes
L_s, M_s, N_s	aerodynamic moments about stability axes
X_b, Y_b, Z_b	total forces along body axes
L_b, M_b, N_b	total moment about body axes
$\dot{u}$, $\dot{v}$, $\dot{w}$	accelerations along body axes
$\dot{p}$, $\dot{q}_1$, $\dot{r}$	rotational accelerations about body axes
u, v, w	velocity along body axes
p, q_1, r	angular velocity about body axes
θ, ψ, ϕ	Euler angles
ℓ_1, m_1, n_1	direction cosines for x-inertial axis
ℓ_2, m_2, n_2	direction cosines for y-inertial axis
ℓ_3, m_3, n_3	direction cosines for z-inertial axis
a	speed of sound
α	angle of attack
b	wing span
β	sideslip angle
C_x	aerodynamic coefficient of drag
C_y	aerodynamic coefficient of side force
C_z, C_L	aerodynamic coefficient lift
C_ℓ	aerodynamic coefficient of roll
C_m	aerodynamic coefficient of pitch
C_n	aerodynamic coefficient of yaw
C. G.	center of gravity

SYMBOLS (continued)

c, M.A.C.	mean aerodynamic chord
DT	drop tanks
DC	drag chute
$DRAG_{WM}$	windmilling drag
δJ	speed brake deflection
δR	rudder deflection
δA	aileron deflection
δH	horizontal stabilizer deflection
d	distance from 35% M.A.C. to center of gravity
ΔM	pitching moment due to nosewheel contact with ground
ϵ, ϵ_2, ϵ_3	correction factors for computation of direction cosines
F_{BR}, F_{BL}	right and left brake force
g	acceleration due to gravity
GE	ground effects
η	normal acceleration
h	pressure altitude
H	altitude above the ground
H	horizontal stabilizer in aerodynamic equations
I_x, I_y, I_z	moments of inertia
K	nosewheel damping factor
LG	landing gear
Ma	Mach number
M_i	instantaneous mass of the aircraft
M_f	mass of fuel
N_2	asymptotic value of RPM

SYMBOLS (continued)

$\%F_n$	percent thrust
q	dynamic pressure
ρ	air density
RPM	engine revolutions per minute
R/C	rate of climb
S	wing area
T	engine thrust
δ	throttle position
V_i	indicated airspeed
V_T	true airspeed
W	wings
WA	air entering the engine inlet duct
W_f	fuel flow

B. COMPUTER TERMS

AC	accumulator
MBR	memory buffer register
LR	live register
XR	index register
PC	program counter
C(y)	contents of register y
→	replaces
s	scale factor
AC_n	bit n of the accumulator
MBR_n	bit n of the memory buffer register
LR_n	bit n of the live register
(x	register whose contents is x

$\bullet+n$	current address plus n

C. MULTIPLIER TERMS

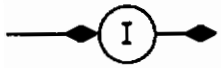
t_{max}	maximum multiply time including memory accesses for an addressable multiply command
t_{avg}	average multiply time including memory accesses for an addressable multiply command
t_{ha}	half-add time
t_{cp}	carry propagate and carry addition time for an n-bit register using conventional carry
t'_{cp}	carry propagate and carry addition time for an n-bit register using high-speed carry
t_s	shift time
t_c	time required to complement a flip-flop
t_{mc}	memory cycle time
n	word length
kn	average maximum carry length for an n-bit word
I	complexity index
N	number of inputs to gates in additional multiplier circuitry
F	number of flip-flops in additional multiplier circuitry

D. LOGIC SYMBOLS

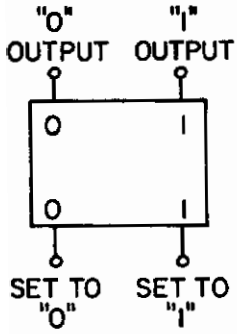
	logic level
	pulse

SYMBOLS (continued)

logic gate whose output will be present if N or more inputs are present



inverter



flip-flop

there is a logical delay built into the flip-flop so that the output may be sensed at the same time that the input is pulsed

CHAPTER I

INTRODUCTION

1.1 Statement of the Problem

Up to the present time, real-time flight simulation has usually been carried out on analog computers. There have been two successful attempts to simulate large-scale aircraft models in real-time on a digital computer.[1, 2*] However, both of these computers were complex and costly.

The object of the investigation reported here is twofold: (1) To test the ability of a moderate-size digital computer, such as the M. I. T. TX-O, to solve a complex, real-time flight simulation problem, and (2) To determine how such a general-purpose digital computer could be modified to facilitate real-time simulation and yet maintain the cost at a level competitive with present analog computers performing the same task.

The particular simulation problem used for this study was the F-100A, a supersonic fighter manufactured by North American Aviation.

In relation to the design of a special-purpose computer, the method of attack in this study was to first determine the exact nature of the F-100A problem. To reduce the labor of data processing and to provide competitive standards, the mathematical model used by Melpar for the analog F-100A simulator and by Sylvania for the UDOFT digital simulator was employed. Then, using this model, the full F-100A problem was programmed with an order code very similar to the TX-O machine language. This program was then a quantitative standard against which the performance of various proposed logical alterations were compared. A functional design of a special-purpose computer gradually evolved that was not only suitable for the specific problem investigated, but also seems applicable to a wide range of real-time simulation problems.

1.2 History of the Problem

In 1955, the Moore School of Electrical Engineering at the University of Pennsylvania[3] completed the design of a digital computer for the simulation of

---

* Superscripts refer to numbered items in the Bibliography.

a supersonic fighter. The construction of this computer, known as the Universal Digital Operational Flight Trainer (UDOFT), was completed in 1960 by Sylvania.[2] The UDOFT computer has been programmed to solve a complete F-100A model at a solution rate of 20 solutions per second.

In August, 1958, work was completed at the M.I.T. Electronic Systems Laboratory[1] on an experimental analog-digital flight simulator. The computations were entirely digital except for the use of peripheral analog integrators. In this study, which was done on the M.I.T. Whirlwind computer, a solution rate of 60 solutions per second was achieved for a simplified set of F-100A equations.

With the exception of these digital studies, all the simulation of aircraft in real-time has been done on analog machines. The cost of either the UDOFT computer or the Whirlwind computer would restrict their use as a replacement for analog computers. However, it is now felt that it would be possible to produce a digital computer at a price that would be competitive with the analog computers now being used for large-scale simulation.

Connelly,[4] at the M.I.T. Electronic Systems Laboratory, has investigated the relationship between cost and computing ability for both digital and analog computing systems. He has hypothesized that digital techniques are competitive with analog techniques for real-time simulation problems approximately the size of the F-100A problem and larger. Since the cost of the analog computer used in the F-100A trainers is known, establishing the size of a digital computer that would solve this same problem provides a means of testing this hypothesis.

The Digital Equipment Corporation's PDP-1 is typical of the commercial digital computers that are now being offered in a price range that makes them competitive with analog techniques for the F-100A problem. Since the M.I.T. TX-O is practically a prototype of the PDP-1, the present study has investigated the ability of the TX-O to solve the full F-100A problem in real-time.

1.3 Outline

After a brief description of the overall problem, Chapter II presents an outline of the F-100A mathematical model, including all of the aerodynamic equations. The order code used to program this model and the results of the program in tabular form are presented in Chapter III. The tables are breakdowns of running time and memory requirements for the TX-O program.

The first of the detailed discussions of particular areas is contained in Chapter IV where the problem of non-linear function generation is analyzed. A description of the methods used by the TX-O and by UDOFT is given and comparisons of accuracy, ease of handling, data storage, and speed are made. Possible revisions for the TX-O order code to speed up non-linear function generation are included. From the results of this chapter and a consideration of the results presented in Chapter III, five instructions not presently included in the TX-O order code, are recommended in Chapter V. The time and memory saved by these instructions is also given.

Since running time was considered the most difficult specification to meet, no subroutines were used in the program described in Chapter III. The use of subroutines is investigated in Chapter VI, and quantitative values of the trade-off between running time and memory are given.

A description of the integration formula used as well as a discussion of the various factors affecting word length are presented in Chapter VII. The word length requirement for the F-100A is derived, and several methods of using machines with shorter word lengths than required by the dynamic range of the variables are presented.

Decisions, inputs and outputs are grouped together in Chapter VIII because discrete inputs are a type of decision and the same additional orders would save time for both. A brief description of the peripheral analog and digital equipment used for handling inputs and outputs is also presented.

Since multiplication consumes a sizable fraction of the operating time, this single order was considered in detail in Chapter IX. A comparison between the running time and difficulty of implementation for several different methods of multiplication is given.

A brief summary of the more important results and recommendations for future work is presented in Chapter X.

CHAPTER II

DESCRIPTION OF THE F-100A MODEL

2.1 Introduction

The mathematical model used for the TX-O simulation program is described in detail in this chapter. Except for aerodynamic function generation and integration, the model is essentially that used by Sylvania in the UDOFT digital simulation program.[2] This in turn was largely based on the Melpar model for the analog OFT, hence is influenced by the rather special limitations of analog implementation. The Sylvania model was used for two reasons. First, one objective of the study was to determine whether commercially-available machines of the TX-O size are adequate for the same problem as that solved by UDOFT. In order to have a basis of comparison, it seemed reasonable that the same model be used. The second reason was that a new analysis of the problem or the analysis of a more modern aircraft would have greatly increased the time and effort required for the present study without adding in a significant way to the value of the analysis.

The model can be divided into several parts which will be briefly described in this chapter:

1) Aerodynamics
2) Altitude
3) Engine
4) Mass, Moments of Inertia
5) Hydraulic Systems
6) Instruments
7) Land-Air-Crash Decisions

Figure 1 shows the main information flow of the program.

DISCRETE INPUTS

START CRANK
START FIRE
EMERGENCY FUEL ON
MAIN FUEL REGULATOR FAILED
AFTERBURNER ON AND NOZZLE FAILED CLOSED
AFTERBURNER OFF AND NOZZLE FAILED OPEN
COCKPIT TEMPERATURE MASTER SWITCH
CANOPY AND WINDSHIELD DEFROST LEVER
WINDSHIELD ANTI-ICE
AFTERBURNER ON
NO FUEL DEPLETION
DROP TANK JETTISON
REFUEL DROP TANKS
DROP TANK PRESSURE
MAIN TANK REFUEL
MAIN TANK DUMP
CENTER OF GRAVITY LOCK
LANDING GEAR IN MOTION
SPEED BRAKE DUMP
SPEED BRAKE IN
SPEED BRAKE OUT
UTILITY HYDRAULIC FAIL
DRAG CHUTE DEPLOYED
TRUE AIRSPEED LOCK
ROLL ANGLE LOCK
AUTOPILOT
NOSE WHEEL STEERING
INCREASE ALTITUDE
DECREASE ALTITUDE
CABIN PRESSURE 2. 75 P. S. I
CABIN PRESSURE 5. 00 P. S. I.
ALTITUDE LOCK
HYDRAULIC SYSTEM NO 1 FAIL
HYDRAULIC SYSTEM NO. 2 FAIL
EMERGENCY HYDRAULIC SYSTEM OPERATING
HYDRAULIC SYSTEM NO. 1 TO ANALOG OUTPUT
HYDRAULIC SYSTEM NO. 2 TO ANALOG OUTPUT
PITOT ICE
ZERO MODE
FREEZE MODE
YAW DAMPER ON
ROUGH AIR
GUIDE VANE ANTI-ICE

ANALOG INPUTS

RIGHT BRAKE FORCE
LEFT BRAKE FORCE
THROTTLE POSITION
AILERON POSITION
HORIZONTAL STABILIZER POSITION
RUDDER POSITION
AIRPORT ELEVATION
BAROMETRIC PRESSURE SETTING

DISCRETE OUTPUTS

DROP TANKS FULL
LANDING GEAR DOWN
LANDING GEAR UP
SPEED BRAKE IN
SPEED BRAKE OUT
SPEED BRAKE IN MOTION
DRAG CHUTE LOST
STALL WARNING
STALL
LAND-AIR
CRASH
HYDRAULIC SYSTEM NO 1 FAIL
HYDRAULIC SYSTEM NO. 2 FAIL
STABILIZER FROZEN
AILERON FROZEN

ANALOG OUTPUTS

DROP TANK FUEL QUANTITY
MAIN TANK FUEL QUANTITY
TAILPIPE TEMPERATURE
INDICATED ALTITUDE
ALTITUDE ABOVE GROUND
CABIN ALTITUDE
HYDRAULIC PRESSURE
NORMAL ACCELERATION
INDICATED AIRSPEED
MACH NUMBER
BALL ANGLE
RATE OF CLIMB
GROUND SPEED
TRUE AIRSPEED
PITCH ANGLE
ROLL ANGLE
TRUE HEADING
RUDDER HINGE MOMENT
ENGINE R. P. M.
TURNING RATE
ROLLING RATE
FUEL FLOW
ANGLE OF ATTACK
ICE QUANTITY

Adapted from Sylvania Final Report FR77-1N

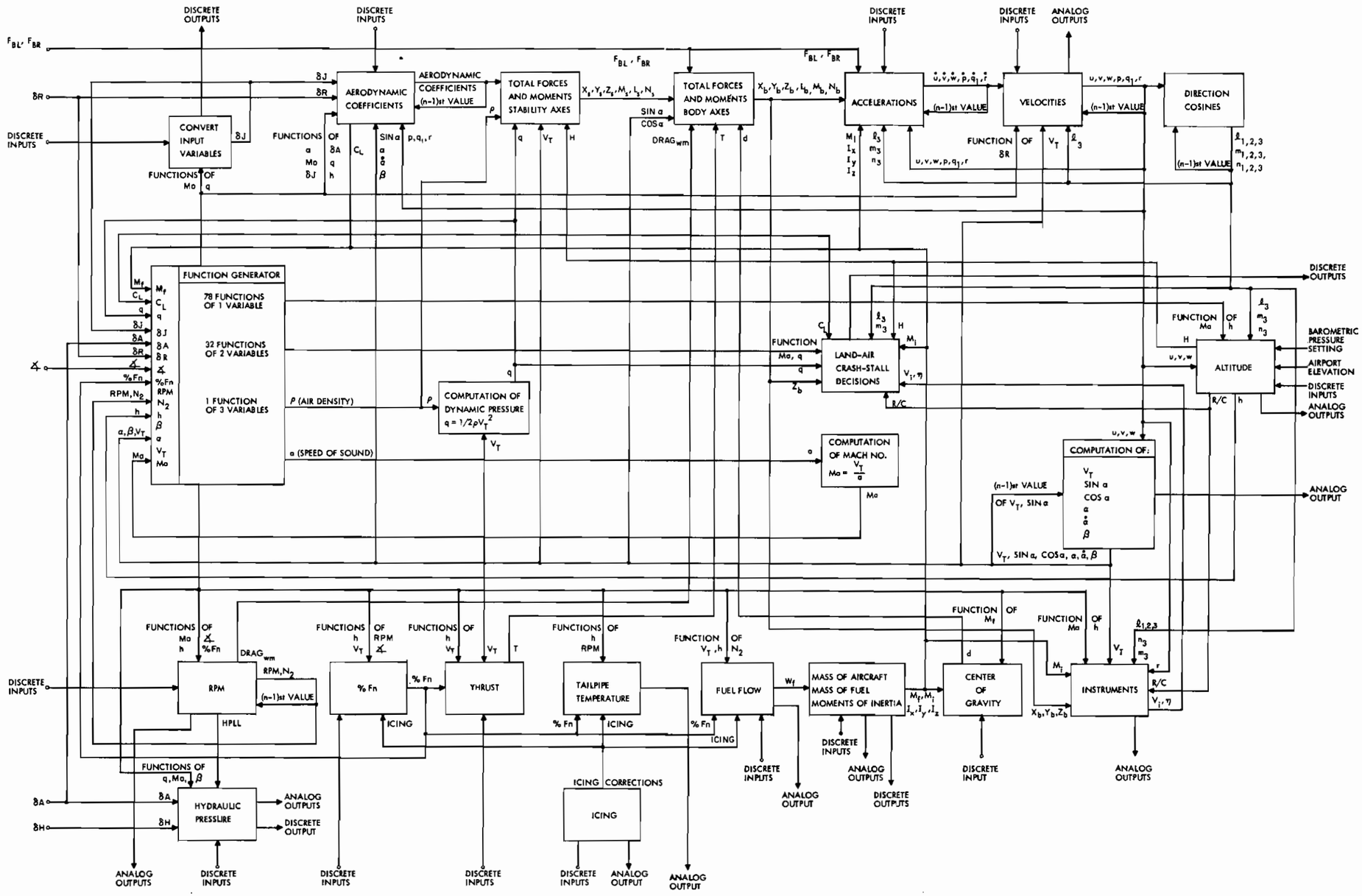


Fig. 1 Main Information Flow

2.2 The F-100A Model

In the F-100A Simulation Program, it is necessary to employ two moving axes systems. These are the body or airplane axis system and the stability axis system, both shown in Figure 2.

The body axis system is a right-handed set of mutually perpendicular axes whose origin is at the aircraft center of gravity. The orientation of the body axes is fixed with respect to the aircraft, but the origin translates slightly in the x axis direction as the center of gravity shifts. The x and z-body axes lie in the plane of symmetry of the aircraft.

The stability axis system is a right-handed set of mutually perpendicular axes whose origin is fixed with respect to the aircraft at the 35% M. A. C. point in the plane of symmetry. Aerodynamic forces and moments are given in terms of this fixed point. The x-stability axis is the projection of the velocity vector of the aircraft on the plane of symmetry. The angle between the velocity vector and the x-stability axis is defined as the sideslip β. The angle between the x-stability axis and the x-body axis is defined as the angle of attack α. The z-stability axis is in the plane of symmetry of the aircraft.

The aircraft manufacturer's aerodynamic data is usually presented in the stability axis system. This data is in the form of non-dimensional aerodynamic coefficients. For an excellent discussion of aerodynamic coefficients and the confusion that results from the lack of standard definitions, see reference 5 in the Bibliography.

The following six equations for the three forces and three rotational moments in the stability axis system are used for the simulation program:

X_s - Total Aerodynamic Force Along the Stability X - Axis

$$X_s = \frac{\rho V_T^2 S}{2} \left[C_x(C_z, Ma) + C_{x(DT)}(Ma) + C_{x(\delta J)}(Ma, \delta J) + C_{x(DC)} + C_{x(LG)} \right] \quad (2.1)$$

Y_s - Total Aerodynamic Force Along the Stability Y - Axis

$$Y_s = \frac{\rho V_T S b}{4} C_{y_p}(\alpha) \cdot (p + r \sin\alpha) + \frac{\rho V_T^2 S}{2} \left[C_{y_\beta}(Ma) \cdot \beta + C_{y_{\delta R}}(Ma, h_p) \cdot \delta R + C_{y(DT)} \right] \quad (2.2)$$

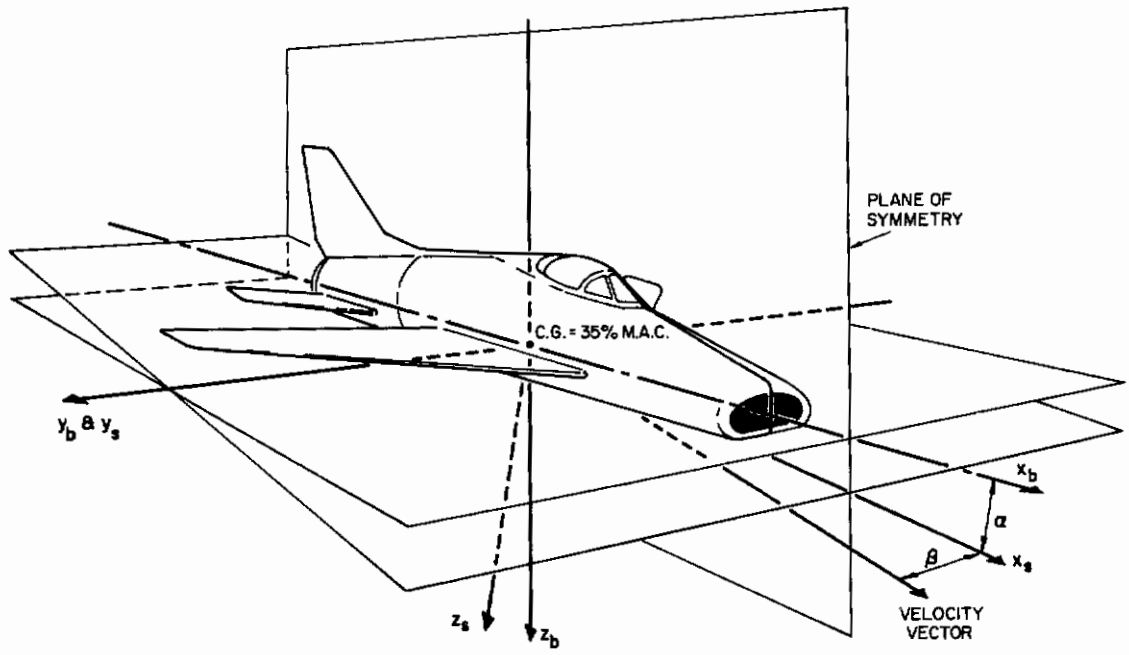


Fig. 2 Definition of Body and Stability Axes

Z_s - Total Aerodynamic Force Along the Stability Z - Axis

$$Z_s = \frac{\rho V_T^2 S}{2} \left[C_{z(W)} (\alpha, Ma, q) + C_{z(H)} (\alpha, Ma, q) + C_{z_{\delta J}} (Ma) \cdot \delta J + C_{z(DT)} (\alpha, Ma) \right] \quad (2.3)$$

L_s - Total Aerodynamic Moment About the Stability X - Axis

$$L_s = \frac{\rho V_T^2 S b}{2} \left[C_{\ell(\delta A)} (\alpha, Ma, q, \delta A) + C_{\ell_{\delta R}} (\alpha, Ma, q) \cdot \delta R + C_{\ell_\beta} (\alpha, Ma, q) \cdot \beta \right] + \frac{\rho V_T S b^2}{4} \left[C_{\ell_p} (\alpha, Ma, q) \cdot (p + r \cdot \sin \alpha) + C_{\ell_r} (\alpha, Ma) \cdot (r - p \cdot \sin \alpha) \right] \quad (2.4)$$

M_s - Total Aerodynamic Moment About the Stability Y - Axis

$$M_s = \frac{\rho V_T^2 S c}{2} \left[C_m (C_z, Ma, q) + C_{m(WA)} (Ma, h, \dot{\alpha}) \cdot \alpha + C_{m(DT)} (\alpha, Ma) + C_{m_{\delta J}} (\alpha, Ma) \cdot \delta J + C_{m(DC)} (\alpha) + C_{m(LG)} (\alpha) + C_{m(GE)} \right] + \frac{\rho V_T S c^2}{4} \left[C_{m_{q_1}} (Ma, h) \cdot q_1 + C_{m_{\dot{\alpha}}} (Ma, h) \cdot \dot{\alpha} \right] + \frac{\rho V_T^2 S b}{2} C_{m(H)} (Ma) \quad (2.5)$$

N_s - Total Aerodynamic Moment About the Stability Z - Axis

$$N_s = \frac{\rho V_T^2 Sb}{2}\left[C_{n_\beta}(Ma, q)\cdot\beta + C_{n_{\delta R}}(Ma, q)\cdot\delta R + C_{n(\delta A)}(\alpha, \delta A) + C_{n(WA)}(Ma, h, \phi)\cdot\beta\right] + \frac{\rho V_T Sb^2}{4}\left[C_{n_p}(\alpha)\cdot(p+r\cdot\sin\alpha) + C_{n_r}(Ma,q)\cdot(r-p\cdot\sin\alpha)\right] \quad (2.6)$$

These three forces and three moments are then transformed into the body-axis system by a simple coordinate transformation. At this point the effect of the engine thrust and windmilling drag are taken into account.

$$X_b = \begin{cases} T + X_s\cos\alpha - Drag_{WM} - Z_s\sin\alpha & \text{(air)} \\ T + X_s\cos\alpha - Drag_{WM} - (F_{BR}+F_{BL}) & \text{(land)} \end{cases} \quad (2.7)$$

$$Y_b = Y_s \quad (2.8)$$

$$Z_b = (0.053)\cdot T + X_s\sin\alpha + Z_s\cos\alpha \quad (2.9)$$

$$L_b = L_s\cos\alpha - N_s\sin\alpha \quad (2.10)$$

$$M_b = M_s - (1.33)\cdot T - Z_s\cdot d \quad (2.11)$$

$$N_b = N_s\cos\alpha + L_s\sin\alpha + Y_s\cdot d \quad (2.12)$$

The total forces along the body axes can then be used to compute the three linear accelerations, $\dot{u}$, $\dot{v}$, and $\dot{w}$, and the three rotational accelerations, $\dot{p}$, $\dot{q}$, and $\dot{r}$:

$$\dot{u} = \begin{cases} \frac{X_b}{M_i} - wq_1 + vr - g\sin\theta & \text{(air)} \\ \frac{X_b}{M_i} - wq_1 + vr & \text{(land)} \end{cases} \quad (2.13)$$

$$\dot{v} = \frac{Y_b}{M_i} + g\cos\theta \sin\phi - ur + wp \tag{2.14}$$

$$\dot{w} = \frac{Z_b}{M_i} + g\cos\theta \cos\phi - vp + uq_1 \tag{2.15}$$

$$\dot{p} = \begin{cases} \dfrac{L_b + (I_y - I_z)q_1 r}{I_x} & \text{(air)} \\ 0 & \text{(land)} \end{cases} \tag{2.16}$$

$$\dot{q}_1 = \begin{cases} \dfrac{M_b + (54,200)rp}{I_y} & \text{(air)} \\ \dfrac{M_b + (54,200)rp - (1250)(g + Z_b/M_i) + \Delta M}{I_y} & \text{(land)} \end{cases} \tag{2.17}$$

ΔM is the pitching moment due to nosewheel contact with the ground.

$$\Delta M = \begin{cases} (25,000)(5 - \theta) - K\dot{\theta} & \theta < 5^\circ \\ 0 & \theta \geq 5^\circ \end{cases} \tag{2.18}$$

$$\dot{r} = \begin{cases} \dfrac{N_b - (41,300)pq_1 + (420)\dot{v}}{I_z} & \text{(air)} \\ \dfrac{N_b - (41,300)pq_1 + (6.21)(F_{BR} - F_{BL}) - (1250)ur + (420)\dot{v}}{I_z} & \text{(land)} \end{cases} \tag{2.19}$$

These six accelerations can then be integrated to get the corresponding velocities:

$$u = \int \dot{u}\, dt \tag{2.20}$$

$$v = \begin{cases} \int \dot{v}\, dt & \text{(air)} \\ 0 & \text{(land)} \end{cases} \tag{2.21}$$

$$w = \begin{cases} \int \dot{w}\, dt & \text{(air)} \\ u \sin \theta & \text{(land)} \end{cases} \tag{2.22}$$

$$p = \int \dot{p}\, dt \tag{2.23}$$

$$q_1 = \int \dot{q}_1\, dt \tag{2.24}$$

$$r = \begin{cases} \int \dot{r}\, dt & \\ f(\delta R)\, V_T & \text{(nosewheel steering)} \end{cases} \tag{2.25}$$

To include the effect of gravity, the orientation of the aircraft with respect to inertial space must be known. A convenient inertia-axis system uses a z-axis in the direction of the earth's gravity vector and an x-axis tangent to the local meridian and pointing in the direction of true north. The orientation of any axis system with respect to the inertia axes can be described by three angles known as Euler angles. The definition of the three Euler angles is shown in Figure 3.

Using the Euler angles for the body axis system, the rate of change of the Euler angles can be computed from the Euler angles and the components of angular velocity.

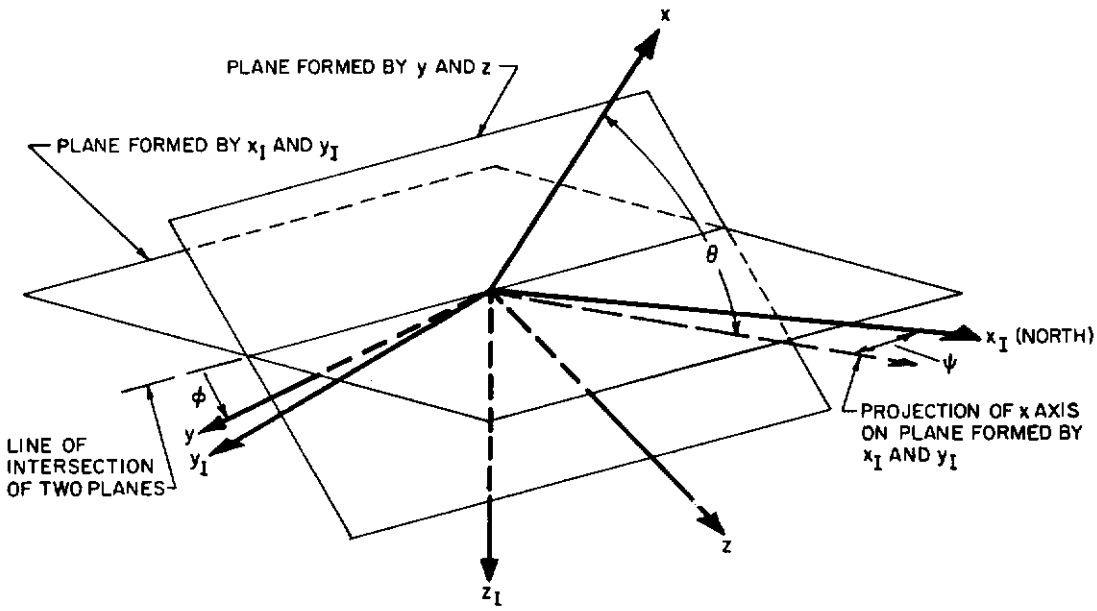


Fig. 3 Definition of Euler Angles

$\ell = \cos \zeta$
$m = \cos \lambda$
$n = \cos \gamma$

SUBSCRIPTS 1, 2, AND 3 ON THE DIRECTION COSINES REFER TO THE x_I, y_I, AND z_I AXES RESPECTIVELY.

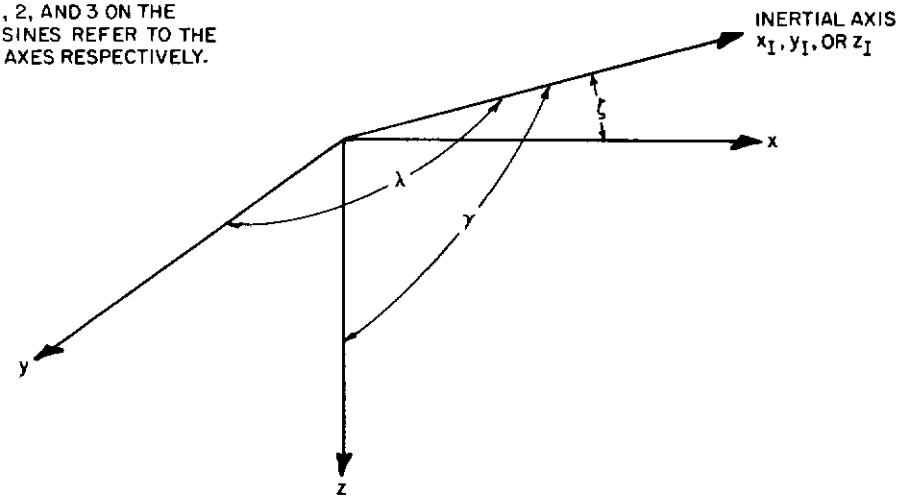


Fig. 4 Definition of Direction Cosines

$$\dot{\theta} = q_1 \cos \phi - r \sin \phi \tag{2.26}$$

$$\dot{\phi} = \frac{1}{\cos \theta} (p \cos \theta + q_1 \sin \theta \sin \phi + r \sin \theta \cos \phi) \tag{2.27}$$

$$\dot{\psi} = \frac{1}{\cos \theta} (q_1 \sin \theta + r \cos \phi) \tag{2.28}$$

Theoretically, these three derivatives could be computed and then integrated to give the three Euler angles. However, in unrestricted simulation, using point-by-point numerical calculation, the equations for $\dot{\phi}$ and $\dot{\psi}$ become indeterminate when θ approaches $\pm 90^{\circ}$.

Consequently, in UDOFT another method is used to describe the orientation of the body axis system in inertial space, the method of direction cosines.

The direction cosines are the components of a transformation matrix transforming the coordinates of a point in one axis system to the coordinates of the point in another axis system which is rotated with respect to the first:

$$\begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} = \begin{bmatrix} \ell_1 & m_1 & n_1 \\ \ell_2 & m_2 & n_2 \\ \ell_3 & m_3 & n_3 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.29}$$

This relationship is also shown in Figure 4.

Direction cosines can be defined in terms of trigonometric functions of the Euler angles:

$$\ell_1 = \cos \theta \cos \psi \tag{2.30}$$

$$m_1 = \sin \theta \sin \phi \cos \psi - \cos \phi \sin \psi \tag{2.31}$$

$$n_1 = \sin \theta \cos \phi \cos \psi + \sin \phi \sin \psi \tag{2.32}$$

$$\ell_2 = \cos \theta \sin \psi \tag{2.33}$$

$$m_2 = \sin \theta \sin \phi \sin \psi + \cos \phi \cos \psi \tag{2.34}$$

$$n_2 = \sin \theta \cos \phi \sin \psi - \sin \phi \cos \psi \tag{2.35}$$

$$\ell_3 = -\sin\theta \tag{2.36}$$

$$m_3 = \cos\theta \sin\phi \tag{2.37}$$

$$n_3 = \cos\theta \cos\phi \tag{2.38}$$

Alternatively, nine differential equations may be used to generate the direction cosines:

$$\dot{\ell}_1 = rm_1 - q_1 n_1 \tag{2.39}$$

$$\dot{\ell}_2 = rm_2 - q_1 n_2 \tag{2.40}$$

$$\dot{\ell}_3 = rm_3 - q_1 n_3 \tag{2.41}$$

$$\dot{m}_1 = pn_1 - r\ell_1 \tag{2.42}$$

$$\dot{m}_2 = pn_2 - r\ell_2 \tag{2.43}$$

$$\dot{m}_3 = pn_3 - r\ell_3 \tag{2 44}$$

$$\dot{n}_1 = q_1\ell_1 - pm_1 \tag{2.45}$$

$$\dot{n}_2 = q_1\ell_2 - pm_2 \tag{2.46}$$

$$\dot{n}_3 = q_1\ell_3 - pm_3 \tag{2.47}$$

There are 21 identities which can be formed from the nine direction cosines:

$$\ell_1^2 + m_1^2 + n_1^2 = 1 \tag{2.48}$$

$$\ell_2^2 + m_2^2 + n_2^2 = 1 \tag{2.49}$$

$$\ell_3^2 + m_3^2 + n_3^2 = 1 \tag{2.50}$$

$$\ell_1^2 + \ell_2^2 + \ell_3^2 = 1 \qquad (2.51)$$

$$m_1^2 + m_2^2 + m_3^2 = 1 \qquad (2.52)$$

$$n_1^2 + n_2^2 + n_3^2 = 1 \qquad (2.53)$$

$$\ell_1\ell_2 + m_1m_2 + n_1n_2 = 0 \qquad (2.54)$$

$$\ell_1\ell_3 + m_1m_3 + n_1n_3 = 0 \qquad (2.55)$$

$$\ell_2\ell_3 + m_2m_3 + n_2n_3 = 0 \qquad (2.56)$$

$$\ell_1m_1 + \ell_2m_2 + \ell_3m_3 = 0 \qquad (2.57)$$

$$\ell_1n_1 + \ell_2n_2 + \ell_3n_3 = 0 \qquad (2.58)$$

$$m_1n_1 + m_2n_2 + m_3n_3 = 0 \qquad (2.59)$$

$$\ell_1 = m_2n_3 - m_3n_2 \qquad (2.60)$$

$$\ell_2 = m_3n_1 - m_1n_3 \qquad (2.61)$$

$$\ell_3 = m_1n_2 - m_2n_1 \qquad (2.62)$$

$$m_1 = n_2\ell_3 - n_3\ell_2 \qquad (2.63)$$

$$m_2 = n_3\ell_1 - n_1\ell_3 \qquad (2.64)$$

$$m_3 = n_1\ell_2 - n_2\ell_1 \qquad (2.65)$$

$$n_1 = \ell_2m_3 - \ell_3m_2 \qquad (2.66)$$

$$n_2 = \ell_3m_1 - \ell_1m_3 \qquad (2.67)$$

$$n_3 = \ell_1m_2 - \ell_2m_1 \qquad (2.68)$$

From this myriad of equations the University of Pennsylvania[6] has developed a very interesting scheme to compute the direction cosines.

The University of Pennsylvania method first computes $\dot{\ell}_2$, $\dot{m}_2$, $\dot{n}_2$, $\dot{\ell}_3$, $\dot{m}_3$, and $\dot{n}_3$ from the differential equations 2.40, 2.41, 2.43, 2.44, 2.46, and 2.47. The (n-1)st values of the direction cosines are used on the right hand side of these equations.

These six derivatives are then integrated to give ℓ_2, m_2, n_2, ℓ_3, m_3, and n_3.

Small drifts in the values of ℓ_3, m_3, and n_3 are corrected by normalizing them using the orthogonalization identity 2.50.

The correction factor, ϵ , is defined by the equation:

$$\epsilon = -1 + (\ell_3^2 + m_3^2 + n_3^2) \tag{2.69}$$

If ℓ_3', m_3', and n_3' are the corrected direction cosines, then the corrected values can be computed from the uncorrected values, ℓ_3, m_3, and n_3, by the equations:

$$\ell_3' = \frac{\ell_3}{\sqrt{1+\epsilon}} \tag{2.70}$$

$$m_3' = \frac{m_3}{\sqrt{1+\epsilon}} \tag{2.71}$$

$$n_3' = \frac{n_3}{\sqrt{1+\epsilon}} \tag{2.72}$$

Since these equations are difficult to use due to the square root and the division, the denominator is expanded around $\epsilon = 0$ and truncated to two terms. The following equations are approximately true for small ϵ :

$$\ell_3' = (1 - \frac{\epsilon}{2})\ \ell_3 \tag{2.73}$$

$$m_3' = (1 - \frac{\epsilon}{2})\ m_3 \tag{2.74}$$

$$n_3' = (1 - \frac{\epsilon}{2})\ n_3 \tag{2.75}$$

If ϵ_3 is defined as $2 - \epsilon$ then these equations further reduce to:

$$\ell_3' = \frac{\epsilon_3}{2} \ell_3 \tag{2.76}$$

$$m_3' = \frac{\epsilon_3}{2} m_3 \tag{2.77}$$

$$n_3' = \frac{\epsilon_3}{2} n_3 \tag{2.78}$$

The ϵ_3 in these equations is computed using the formula:

$$\epsilon_3 = 2 - \epsilon = 2 - \left[-1 + (\ell_3^2 + m_3^2 + n_3^2) \right] \tag{2.79}$$

$$\epsilon_3 = 3 - (\ell_3^2 + m_3^2 + n_3^2) \tag{2.80}$$

The procedure used by the University of Pennsylvania is to compute ϵ_3 using equation 2.80. This value is then used in equations 2.76, 2.77, and 2.78 to compute values of the corrected direction cosines, ℓ_3', m_3', and n_3'.

Now that ℓ_3', m_3', and n_3' are orthogonalized they may be used to orthogonalize ℓ_2, m_2, and n_2 with respect to ℓ_3', m_3', and n_3'. This is done using identity 2.56. This equation can be solved for either ℓ_2, m_2, or n_2. The equation for one of these direction cosines will have either ℓ_3', m_3', or n_3' respectively in the denominator of the right hand side. In order to avoid a zero denominator, the largest of ℓ_3', m_3', or n_3' is chosen. The equation used, assuming for example that n_3' is the largest, would be:

$$n_2^* = \frac{-\ell_2 \ell_3' - m_2 m_3'}{n_3'} \tag{2.81}$$

Next, this set, ℓ_2, m_2, and n_2^*, is orthogonalized using identity 2.49 in the same way as ℓ_3, m_3, and n_3:

$$\epsilon_2 = 3 - (\ell_2^2 + m_2^2 + n_2^{*2}) \tag{2.82}$$

$$\ell_2' = \frac{\epsilon_2}{2}\ \ell_2 \tag{2.83}$$

$$m_2' = \frac{\epsilon_2}{2}\ m_2 \tag{2.84}$$

$$n_2' = \frac{\epsilon_2}{2}\ n_2^* \tag{2.85}$$

Now the last three direction cosines, ℓ_1, m_1, and n_1, are computed making use of equations 2.60, 2.63, and 2.66:

$$\ell_1 = m_2'n_3' - m_3'n_2' \tag{2.86}$$

$$m_1 = n_2'\ell_3' - n_3'\ell_2' \tag{2.87}$$

$$n_1 = \ell_2'm_3' - \ell_3'm_2' \tag{2.88}$$

Since ℓ_1, m_1, and n_1 are computed using direction cosines that are orthogonalized, the set ℓ_1, m_1, and n_1 is also orthogonalized both with respect to each other and with respect to the other direction cosines.

This completes the aerodynamic part of the problem. Next, the new values of the feedback parameters are calculated from their present values and the velocities:

$$V_T(n) = \frac{1}{2}\left[V_T(n-1) + \frac{u^2 + v^2 + w^2}{V_T(n-1)} \right] \tag{2.89}$$ *

$$\sin\ \alpha = \frac{w}{V_T} \approx \alpha\ (\text{rad.}) \tag{2.90}$$

$$\cos\ \alpha = \frac{u}{V_T} \tag{2.91}$$

$$\tan\ \beta = \frac{v}{u} \approx \beta\ (\text{rad.}) \tag{2.92}$$

---

* This is an approximation to the formula $V_T(n) = \sqrt{u^2 + v^2 + w^2}$. For a derivation of this square root algorithm see Reference 7 in the Bibliography.

$$\text{Mach} = \frac{V_T}{a} \tag{2.93}$$

$$\dot{\alpha} = \frac{\sin \alpha\,(n) - \sin \alpha\,(n-1)}{\Delta t} \tag{2.94}$$

In the next routine, the various altitudes used by the program are computed. First, the two main altitudes, which are pressure altitude and true altitude above the ground, are computed. Both of these altitudes are computed by integrating the rate of climb on each cycle and taking into account original airport elevation and present ground elevation. The pressure altitude is then modified by a non-linear function of Mach no. and altitude to give the indicated pressure altitude, which is then converted to an analog voltage and sent to the pilot's altimeter.

The engine parameters, which are computed next, are second only to the aerodynamic equations in complexity. In the engine computations, engine speed, windmilling drag, thrust, fuel flow, and tailpipe temperature are computed as non-linear functions of Mach no., altitude, and pilot throttle setting. In addition, all of these parameters are subjected to a first -order time lag to simulate the few seconds delay time between pilot action and engine response.

The instructor, through his console, is able to specify any one of several icing rates. The effect of icing is to decrease the thrust and increase the fuel flow and tailpipe temperature. The icing routine performs the necessary calculations.

Next the mass of fuel in the tanks, total mass of the aircraft, location of the center of gravity, and moments of inertia are computed. The moments of inertia about the x and z axes are functions of the mass of fuel; the moment of inertia about the y axis is considered constant.

The F-100A has two independent hydraulic systems as well as an emergency system. The average hydraulic pressure is computed and compared with the pressure required to actuate the control surfaces of the aircraft. If the available pressure is insufficient, the pilot's controls, such as the stick and rudder, are frozen in place. The instructor can cause either or both systems to fail in order to test whether the pilot knows the correct emergency procedures.

The instrument calculations are the next routine in the program. This routine prepares the outputs to the pilot's and instructor's instruments.

Typical instruments are normal acceleration, indicated airspeed, Mach number, ball angle, rate of climb, ground speed, and gyro-horizon. These calculations are complicated by the fact that most of the instruments are not linear. The analog voltage necessary to drive the meters in usually a non-linear function of the variable.

The last routine of the simulation model used is the land-air-crash decisions. In this routine the program decides whether the aircraft is in the air, on the ground, about to stall, stalled, or crashed. The proper discrete outputs to the instructor are set accordingly.

CHAPTER III

DESCRIPTION OF THE TX-O FLIGHT SIMULATION PROGRAM

A substantial amount of time was spent in programming the full F-100A problem in a language very similar to the TX-O language. The reasons for expending this effort are threefold: 1) To gain familiarity with the specific digital operations that must be carried out to effect a solution of the complete aircraft model. This information has been analyzed and used as a guide for recommending improvements in the TX-O capabilities. 2) To verify the fact that a computer of the TX-O class is adequate to solve the F-100A problem on a real-time basis. 3) To have a quantitative standard available against which to compare the performance of any proposed logical alterations.

It is necessary at this point to elaborate on the instructions used for this all-digital program. With the exception of four orders, the instructions used are those that are either operating now or are scheduled for installation on the TX-O in the late summer or fall of 1961[8] A brief description of the TX-O, as well as a list of the instructions to be available, is presented in Appendix I. The four instructions not included in this list are also scheduled for eventual inclusion in the TX-O order code, although not with the execution times shown. The instructions are:

mpy multiply	fraction multiply the contents of the accumulator by the contents of the live register, leaving the product in the accumulator.	execution time 25 μsec.
dvf divide	fraction divide the contents of the accumulator by the contents of the live register, leaving the quotient in the accumulator.	execution time 40 μsec.

ars n accumulator right shift n places	shift the contents of the accumulator right n places.	execution time 12 μsec.
als n accumulator left shift n places.	shift the contents of the accumulator left n places.	execution time 12 μsec.

The high-speed multiply command is absolutely necessary for an all-digital simulation program because of the large number of multiply operations. The shift right and shift left commands are required by the type of scaling used in the problem. The divide command is necessary, but the importance of the speed of the command is open to question since only a few divide operations are used.

Since one of the primary purposes of this study was to verify the ability of the TX-O to solve the same problem as was solved by UDOFT and since the UDOFT flow charts were available,[2] it was decided to use the exact UDOFT model for most of the program. The TX-O procedure did deviate from the UDOFT procedure in two important aspects. The first, and most important deviation was in the method of aerodynamic function generation. The UDOFT model used sums and products of functions of one variable to simulate functions of more than one variable. For the TX-O program the original North American Aviation Data was used.[9] Therefore, if an aerodynamic coefficient appeared as a function of two variables in the manufacturer's data, then the TX-O program generated that function of two variables. More will be said about the relative advantages and disadvantages of both of these methods in Chapter IV on Function Generation. Function generation was accomplished on the TX-O by linear interpolation between stored discrete points.

The method of integration used by the TX-O program also differed from the method used by UDOFT. UDOFT used a quadrature formula developed at the University of Pennsylvania called Mod Gurk.[10] This method uses the

past three values of the variable plus the past three values of the derivative. The TX-O method uses only the past value of the variable plus the last two values of the derivative. The experimental justification for using this trapezoidal integration formula is presented in Chapter VII.

The complete TX-O program is too lengthy to incorporate in this document, but the results of an analysis of the program are summarized in Tables I through IV. Table I gives the normal and worst-case timing for each individual routine of the program. The normal timing is the amount of time that a given routine would require on the average with no aircraft failures and with none of the variables approaching their maximum value. The worst-case timing corresponds to that set of conditions which result in maximizing the amount of time required to go through a particular routine. The worst-case path of any particular routine is chosen without consideration for the conditions required for the worst-case path of another particular routine. In fact, the worst-case paths of two routines are very often quite contradictory. The worst case of one routine might require the aircraft to be on the ground with nosewheel steering employed while the worst case of another routine might require the aircraft to be going to a speed greater than Mach 1.1.

Table II gives the breakdown of worst-case timing by operations. Table III gives the breakdown of memory requirements by routines. Table IV gives the breakdown of memory requirements by operations.

Note that Table IV often contains more routines of a particular type than does Table II. For example, Table IV lists 78 functions of one variable in memory while Table II lists only 48 functions of one variable in the worst-case timing. This is due to the fact that in many places in the program there are alternate paths both of which contain separate functions of one variable. The number of functions in the worst-case timing includes only those functions in one path while the total memory requirement includes all the functions in both paths.

It should also be noted that many operations in Tables II and IV are included in more than one table entry. For example, the multiplication instructions used for function generation are listed twice, once under function generation, and once under multiply commands.

The total of the worst-case times for each routine is 53.915 milliseconds which means that under these rather improbable circumstances the

program could not be run on the TX-O at twenty solutions per second. The various logical improvements suggested by the program are treated in detail in the following chapters. Estimates of the savings in time and memory are cited for each improvement suggested.

TABLE I

Program Timing

Routine	Normal Running Time in μsecs.	Worst-Case Running Time in μsecs.
Convert Input Variables	1797	2163
Aerodynamic Coefficients	15499	15883
Total Forces and Moments - Stability Axes	1102	1251
Total Forces and Moments - Body Axes	1188	1188
Accelerations	2189	2945
Velocity Vectors	1606	1906
Direction Cosines	4074	4074
Mach, Dynamic Pressure, Airspeed, etc.	1734	1998
Altitude	2808	2988
Engine RPM	1669	1813
Percent Thrust	1000	1431
Thrust	1558	1682
Icing	479	509
Fuel Flow	2186	2336
Tailpipe Temperature	1069	1247
Mass of Fuel	613	1177
Mass, Center of Gravity, and Moments of Inertia	1022	1022
Hydraulic System	3206	3398
Instruments	3545	3758
Decisions	408	714
Governing Control	432	432
TOTAL	49184	53915

TABLE II

Breakdown of Worst-Case Timing by Operations

Type of Operation	No.	Time (μsec)	%of Worst-Case Time
Level Select and Slope Generation	13	4420	8.2
Generation of Functions of 1 Variable	48	5232	9.7
Generation of Functions of 2 Variables	31	9031	16.7
Generation of Functions of 3 Variables	1	708	1.3
Miscellaneous Operations		4334	8.0
Total for Function Generation		23715	44.0
Decisions excluding Discrete Inputs	130	4422	8.2
Discrete Inputs	51	1680	3.1
Total for Decisions	182	6102	11.3
All Multiply Commands (25 μsec.)	393	9825	18.2
Analog Outputs	29	1392	2.6
All Division Commands (40 μsec.)	33	1320	2.4
Integrations	12	1236	2.3
Analog Inputs	8	576	1.1
Discrete Outputs	9	432	0.8

TABLE III

Memory Requirement

Routine	Instructions	Data
Convert Input Variables	239	94
Aerodynamic Coefficients	1136	1934
Total Forces and Moments - Stability Axes	91	13
Total Forces and Moments - Body Axes	91	8
Accelerations	226	24
Velocity Vectors	234	45
Direction Cosines	294	38
Mach, Dynamic Pressure, Airspeed, etc.	190	43
Altitude	300	88
Engine RPM	290	94
Percent Thrust	120	61
Thrust	215	102
Icing	42	6
Fuel Flow	356	133
Tailpipe Temperature	128	70
Mass of Fuel	102	15
Mass, Center of Gravity, and Moments of Inertia	67	29
Hydraulic System	302	112
Instruments	274	85
Decisions	95	32
Governing Control	90	8
	4882	3034
Total Instruction and Data		7916

TABLE IV

Memory Requirement Breakdown

Type of Operation	No.	Registers Used	% of Total Memory Required by Problem
Instructions	4882	4882	61.7
Function Table Storage		2274	28.7
Breakpoint Table Storage	15	91	1.1
Parameters and Constants		669	8.5
Breakdown of Instructions by Operations			
Level Select and Slope Generation	15	285	3.6
Generation of Functions of 1 Variable	78	624	7.9
Generation of Functions of 2 Variables	32	672	8.5
Generation of Functions of 3 Variables	1	51	0.6
Miscellaneous Operations		276	3.5
Total Function Generation		1908	24.1
Decisions Excluding Discrete Inputs	152	467	5.9
Discrete Inputs	57	171	2.2
Total Decisions	209	638	8.1
Integrations	12	131	1.7
Analog Outputs	29	116	1.5
Discrete Outputs	15	91	1.1
Analog Inputs	8	56	0.7

CHAPTER IV

FUNCTION GENERATION

4.1 Introduction

As can be seen from Table II function generation is the most time-consuming operation in the F-100A program, requiring 44% of the unmodified worst-case running time. The memory used for function generation is substantial also, 24.1% for function generation instructions and 29.8% for function and breakpoint tables.

Non-linear functions occur mainly in four different places in the program:

1) Aerodynamic coefficients.
2) Engine computations.
3) Hydraulic system.
4) Instruments.

The method of handling aerodynamic coefficients on the TX-O differed from that used by UDOFT. It is felt that the method used for the TX-O is more versatile and accurate. In order to make a meaningful comparison, however, it is necessary to go into detail on both methods.

4.2 Description of the UDOFT Function Generation Method

The analysis of non-linear functions used by UDOFT was actually done originally by Melpar. Melpar manufactured a number of analog trainers for the F-100A, and their method of handling function generation was naturally a method well suited to analog computation. Functions of more than one variable are difficult to generate on analog computers, so Melpar reduced functions of more than one variable into products and sums of functions of one variable.

As a specific example, the coefficient of pitch due to speed brake deflection will be used. In Figure 5 a, b, and c, three UDOFT functions, $f_{20}(Ma)$, $f_{21}(Ma)$, and $f_2(\alpha)$, are shown.[11] UDOFT calculated this coefficient using the formula:

$$C_{m_{\delta J}} = f_{20}(Ma) + f_{21}(Ma) \cdot f_2(\alpha) \qquad (4.1)$$

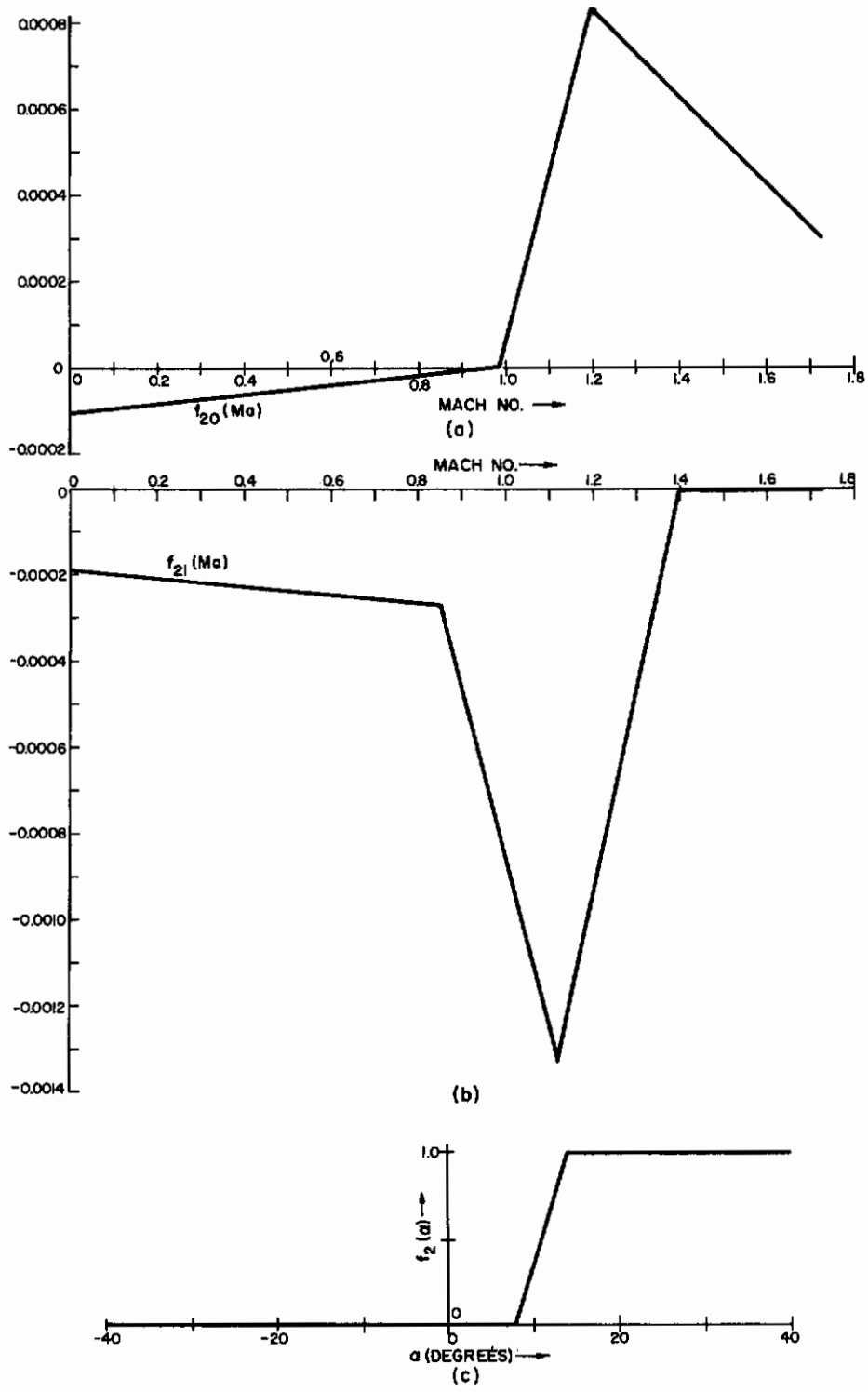


Fig. 5 Functions of a Single Variable Used by UDOFT for $C_{m_{\delta J}}$

This computation has been performed for several values of Mach No., and the results are shown in Figure 6. The function plotted is actually $C_{m_{\delta J}} \cdot \delta J_{max}$ where δJ_{max} is the maximum speed brake deflection, which on the F-100A is 50°.

By way of comparison, Figure 7 shows the original aircraft manufacturer's data.[9] The function plotted is also $C_{m_{\delta J}} \cdot \delta J_{max}$ and the scales are the same as in Figure 6.

It can be seen that the UDOFT (originally, Melpar) data in Figure 6 represents a linearized form of the aircraft manufacturer's data in Figure 7. Whether the UDOFT representation of this coefficient is adequate or inadequate is a matter for an aerodynamicist to decide. The only purpose for making the preceding comparison is to point out the relative advantages and disadvantages of the UDOFT method.

Certain characteristics of the UDOFT method are apparent from a study of Figures 5, 6, and 7. First, the component functions of a single variable bear little relation to the composite function of two variables, particularly the functions of Mach No. Secondly, it is not at all obvious, using this method, how one would go about synthesizing the function of two variables to a higher degree of accuracy. It might be done by increasing the number of breakpoints of the component functions or it might be necessary to increase the number of component functions. If additional functions were used, the original functions of one variable might also have to be changed. Clearly, there is a considerable amount of data analysis necessary in going from the piecewise-linear representation of a function of more than one variable to the component functions of one variable used to generate it.

Another aspect of the UDOFT analysis is that going from the manufacturer's aircraft data to the piecewise-linear function of two variables is also difficult. It is obvious that the allowable aerodynamic approximations have been taken into account in going from Figure 7 to Figure 6. One such factor might be the possible range of values of angle of attack corresponding to a given value of Mach No.

The method of storage and computation of the functions of one variable in UDOFT differs from that used by the TX-O program. As an example, consider $f_{20}(Ma)$ reproduced in Figure 8 for convenience.

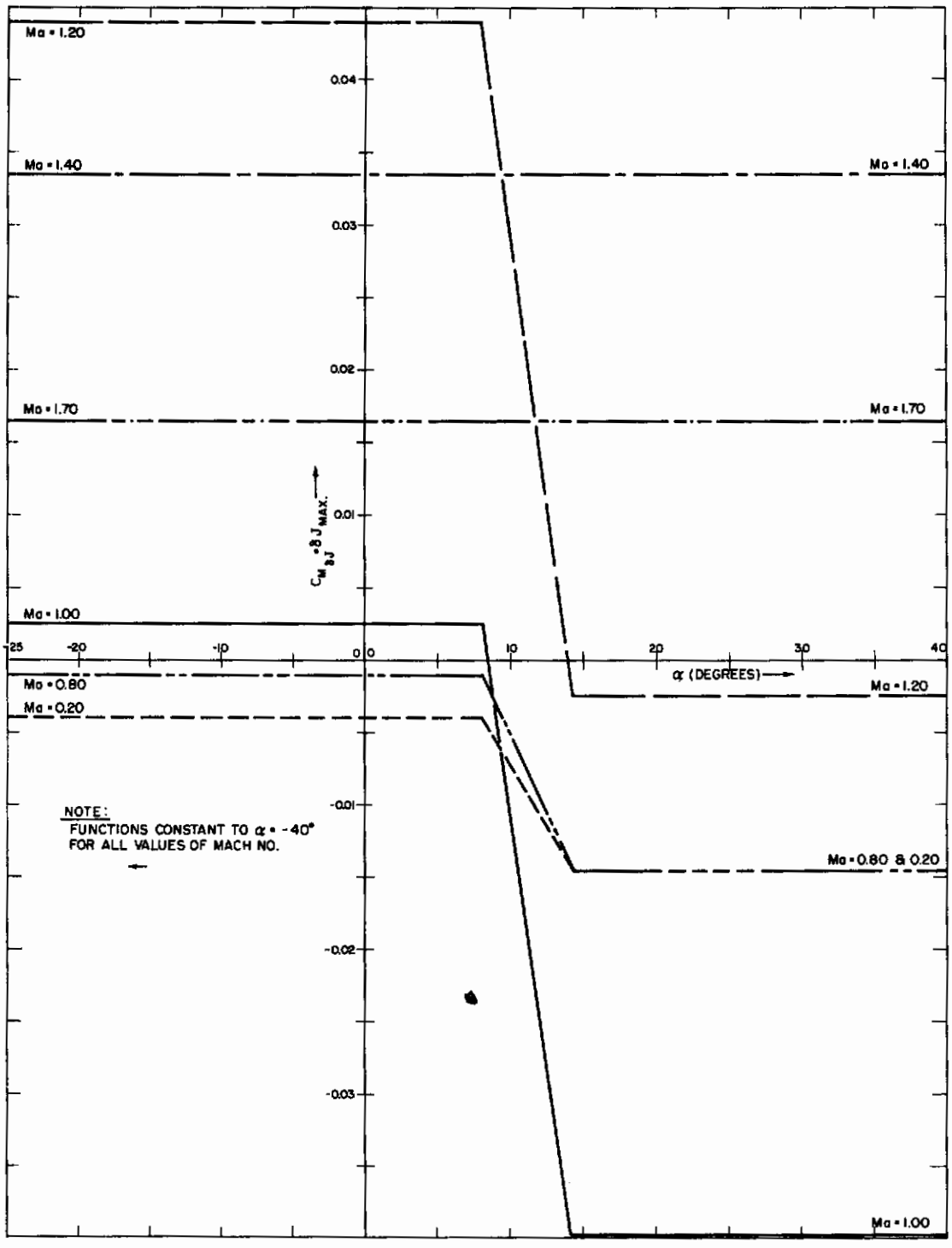


Fig. 6 $C_{m_{\delta J}}$ Used by UDOFT as a Function of Two Variables

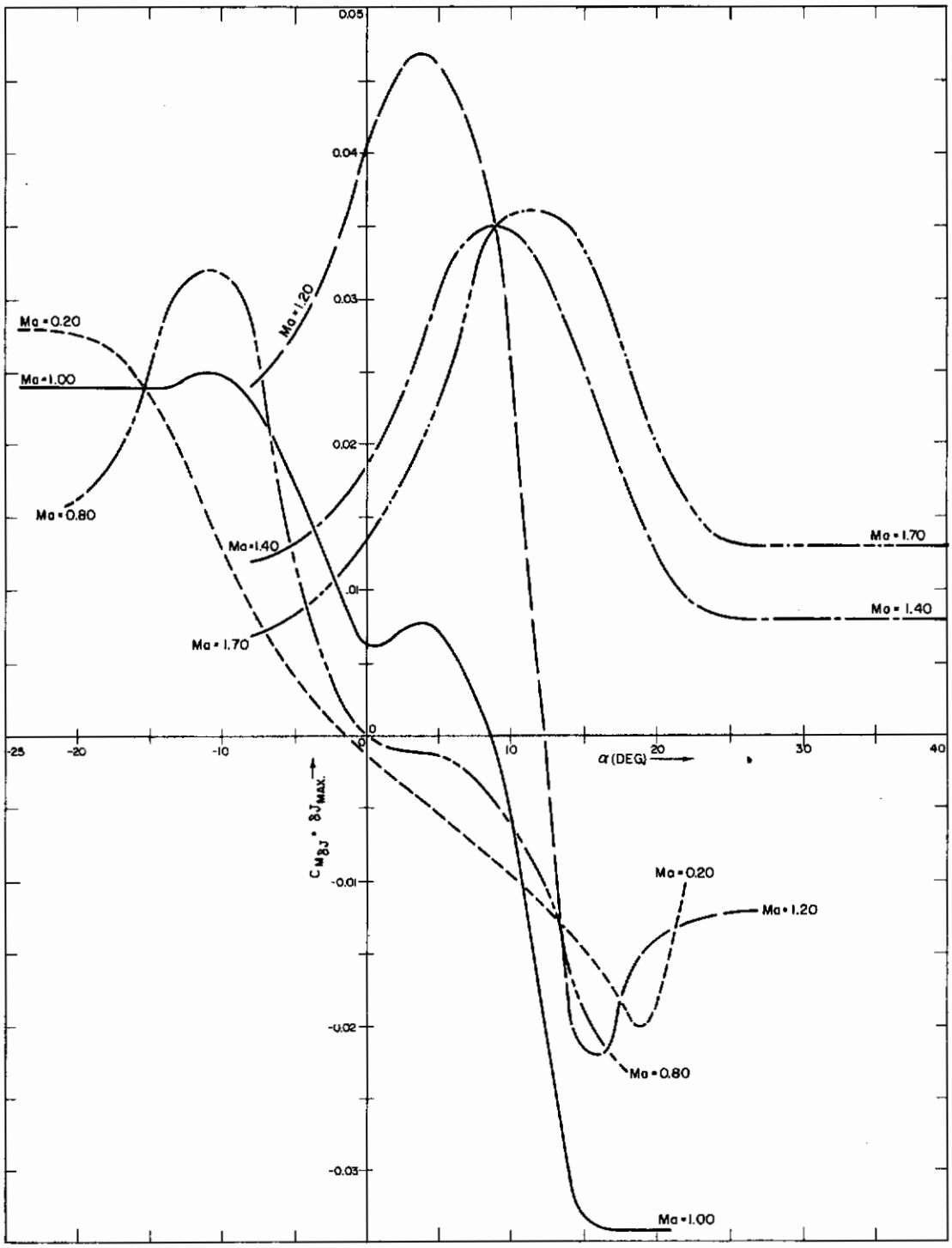


Fig. 7 Aircraft Manufacturer's Data for $C_{m_{\delta J}}$

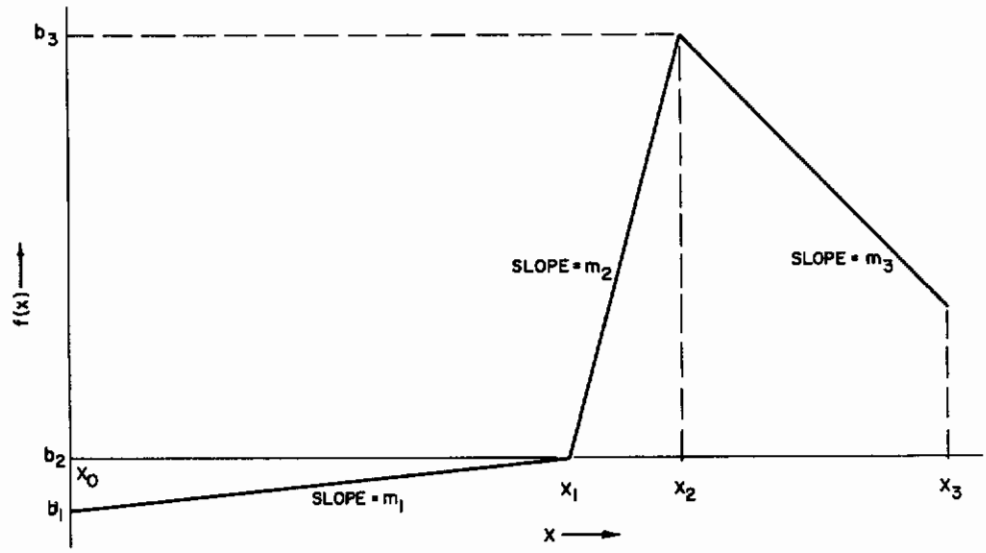


Fig. 8 Notation Used for Describing UDOFT Function Storage

Referring to the notation used in Figure 8 the table storage for $f_{20}(Ma)$ in UDOFT would be:

x_0

m_1

b_1

x_1

m_2

b_2

x_2

m_3

b_3

x_3

It can be seen that a function with n breakpoints would require 3n-2 registers of storage.

Assuming that the independent variable x is in the i^{th} zone, i.e., $x_i < x \leqslant x_{i+1}$, the value of $f_{20}(x)$ would be computed by the formula:

$$f_{20}(x) = b_{i+1} + (x - x_i)\, m_{i+1} \tag{4.2}$$

There are two significant points about the UDOFT method of function storage. First, each individual function of one variable carries its own breakpoint table. $f_{21}(Ma)$ need not have the same breakpoints in Mach No. as does $f_{20}(Ma)$, and, in fact, an investigation of Figures 5a and 5b shows that the breakpoints are different. This is both an advantage and a disadvantage. The advantage lies in the versatility of independently specifying the breakpoints of each function. The disadvantage lies in the extra table storage and the extra computation time required. The extra computation time is involved in an operation, which, for the purpose of this study, will be called level selecting. Level selecting is the operation of determining between which two breakpoints the present value of a variable lies. In the UDOFT method, level selecting must be carried out for each function. More will be said about level selecting later in this chapter, since it is also an important part of the TX-O method of function generation.

The second significant point about the UDOFT function storage is that the slopes, $m_1, m_2, \ldots, m_n$, are also stored. The value of any slope represents redundant information since any particular slope can be computed using other information that is given in the table:

$$m_i = \frac{b_{i+1} - b_i}{x_i - x_{i-1}} \tag{4.3}$$

The advantage of using additional memory to store slopes lies in the fact that the computation proceeds more rapidly. The UDOFT computer has a relatively slow divide instruction (105 μsec.),[12] and the extra time to calculate the slope for each function would be considerable.

4.3 Description of the TX-O Function Generation Method

In contrast to the UDOFT method, the analysis of the manufacturer's data required for the TX-O method for the illustrative coefficient $C_{m_{\delta J}}$ consists simply of selecting breakpoints for α and Mach no. and in reading off and

storing the values of $C_{m_{\delta J}}$ for these particular points. If, in general, there are n breakpoints in α and m breakpoints in Mach no., then the table of values of the function at these discrete points will be m times n registers long. As an example, Figure 9 shows a linearized form of $C_{m_{\delta J}}$ obtained by using eight breakpoints in α and six breakpoints in Mach no. The encircled points would be stored in a table which in this example would be 48 registers long.

In computing the value of a function of two variables, level selecting is first carried out for both variables. The tables of breakpoints are stored separately from the function table in the form given below, which requires an additional n+m registers.

x_0
x_1
x_2
.
.
.
x_{n-1}

} n breakpoints for the variable x

y_0
y_1
y_2
.
.
.
y_{m-1}

} m breakpoints for the variable y

In the TX-O method, no slopes are stored. When the level selecting is completed for a particular variable, x, the output is not only the zone i, such that $x_i < x \leqslant x_{i+1}$, but also the value of the ratio, $(x-x_i)/(x_{i+1}-x_i)$. This value is stored for use by the part of the program that actually computes the interpolated value of the function.

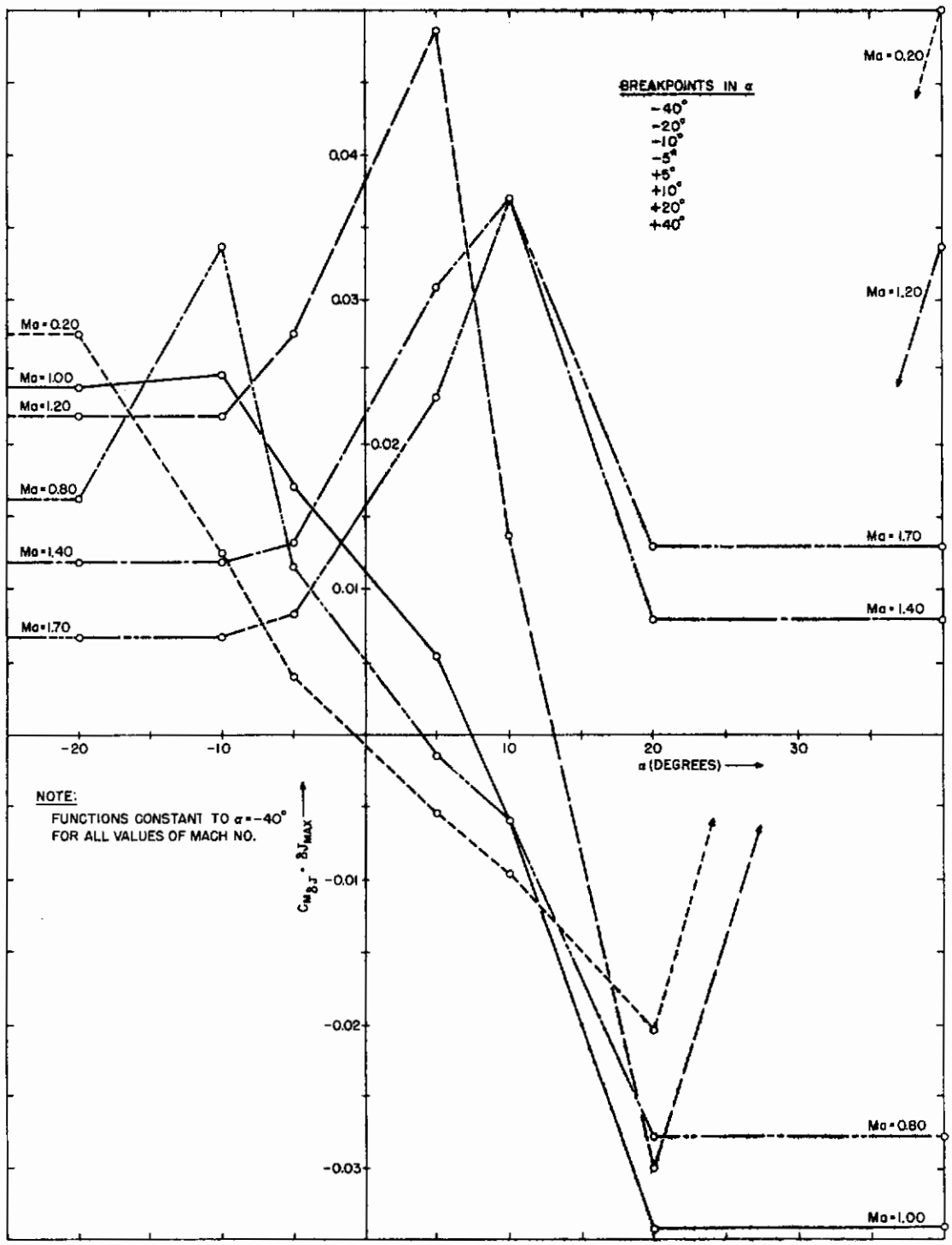


Fig. 9 $C_{m_{\delta J}}$ Used by TX-0 as a Function of Two Variables

The notation to be used in describing functions of two variables for the TX-O program is that the value of the function at the point $x=x_i$ and $y=y_j$ is $f_{i,j}$. The method of generating a function of two variables, x and y, assuming that x is in zone i and y is in zone j, is a linear interpolation in two variables:

$$f_{x,y} = f_{x,j} + (f_{x,j+1} - f_{x,j}) \frac{y-y_j}{y_{j+1}-y_j}$$

where

$$f_{x,j} = f_{i,j} + (f_{i+1,j} - f_{i,j}) \frac{x-x_i}{x_{i+1}-x_i} \qquad (4.4)$$

$$f_{x,j+1} = f_{i,j+1} + (f_{i+1,j+1} - f_{i,j+1}) \frac{x-x_i}{x_{i+1}-x_i}$$

This method is shown graphically in Figure 10. Notice no divisions are required in this computation since $(x-x_i)/(x_{i+1}-x_i)$ and $(y-y_j)/(y_{j+1}-y_j)$ were computed and stored by the level select routine.

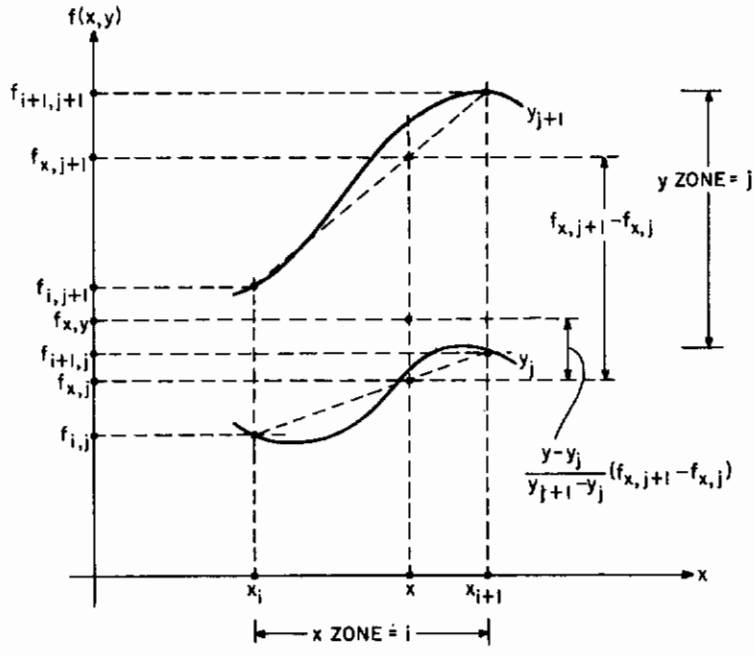


Fig. 10 Two Variable Linear Interpolation

Before discussing memory requirements and program time, some comments on the accuracy of the TX-O method of function generation are in order. It can be seen that for a continuous function of two variables, the accuracy can be increased by simply increasing the number of breakpoints. From an examination of Figures 5 and 6 it is clear that with four breakpoints in α and seven breakpoints in Mach no. the two different methods would yield identical results. By taking a few more breakpoints in α, the TX-O method could be made much more accurate than the UDOFT method.

Thus, as far as accuracy is concerned, the TX-O method is easily extended in accuracy by the use of additional breakpoints, whereas extending the accuracy of the UDOFT method involves a much more complicated analytical procedure.

Another aspect of the accuracy problem is the modification of functions after the program is running. Often the aircraft manufacturer issues revised aerodynamic data while the aircraft is under development, and even after the aircraft is operational, as more reliable information becomes available. Other changes in aerodynamic data result from major modifications in the aircraft. The TX-O method minimizes the difficulty in incorporating the modifications into the program. Usually the same breakpoints can be used so that all that is necessary is the reading off of new values of the function from the new data and changing the function table in the program accordingly. Incorporating modifications into the UDOFT method is a much more difficult task. Quite probably all of the original data analysis would have to be repeated for the particular coefficient altered.

The reason for disassociating the level-selecting routine from the function generation routine in the TX-O is that the same breakpoint table is used for more than one function. It is possible, if enough breakpoints are used for a particular variable, to use only one set of breakpoints for every function that depends on that variable. For the TX-O program, each of the fifteen independent variables used in function generation has only one breakpoint table of no more than eight breakpoints. This method is admittedly less versatile than the UDOFT method of associating a breakpoint table with each function; however, there is no inherent reason why the TX-O program could not use more than one breakpoint table for any particular variable. In the extreme, the TX-O program could have a breakpoint table for each variable for each function; however, this extreme should never be necessary. In addition, there is

no reason to limit the number of breakpoints for a particular variable to eight. However, it should be realized that as the number of breakpoints for a particular variable increases, the size of all the function tables of the functions that depend on that variable also increase by the same factor. For example, assume that the function tables of all the functions that depend on a now require 1000 registers and the number of breakpoints in a is ten. Then if the number of breakpoints in a was increased to fifteen, keeping the same number of breakpoints for the other variables, the function tables would require 1500 registers.

If the number of breakpoint tables for a particular variable were to increase, the greater amount of time required by level selecting would be a factor in considering the advisability of such an increase.

In the TX-O program, level selecting for a single independent variable and generating the ratio, $(x-x_i)/(x_{i+1}-x_i)$, requires a maximum of 340 μsec. This times the thirteen independent variables encountered in the worst-case timing yields the 4.420 msec. figure given in Table II. This assumes only one breakpoint table for each variable. The figure of 340 μsec. for level selecting does not depend on the number of breakpoints in the table for the assumption is made that the variable lies in the same zone it was in during the previous solution cycle or in one of the zones on either side of this zone. If a method is used that does not depend on a knowledge of the previous zone, then level selecting by a linear search routine would require, in the worst case, 188 μsec. + 48n μsec. where n is the number of breakpoints. If n is equal to 8, then this method would require a maximum of 572 μsec. If n is very much larger, the time required becomes prohibitive.

From the preceding discussion it is apparent that an additional 340 μsec. would be required in the worst case for each additional breakpoint table.

4.4 Comparison of Memory Requirements

In order to compare the relative merits of the UDOFT and TX-O function generation techniques, the computation time and memory requirement for the generation of the F-100A aerodynamic coefficients were analyzed for each approach.

The independent variables used for the TX-O aerodynamic function generation and the number of breakpoints associated with each variable are listed in Table V.

Variable	Number of Breakpoints
α	8
M_a	8
C_L	8
δJ	6
h	5
δA	5
q	4

Table V Number of Breakpoints Used in the TX-O Program for Variables Used in Aerodynamic Coefficients

All variables use eight breakpoints except where the form of the manufacturer's data makes this impossible. The total aerodynamic function storage and breakpoint storage requirement for the TX-O approach is 1675 registers. This is made up of one function of three variables, twenty-eight functions of two variables, and fifteen functions of one variable. The average number of breakpoints for the TX-O method is 6.9 breakpoints per function For the UDOFT method there are 1000 registers required for eighty functions of one variable, with an average of 4.8 breakpoints per function. This information is summarized in Table V I.

	UDOFT	TX-O
Functions of 1 Variable	80	15
Functions of 2 Variables	-	28
Functions of 3 Variables	-	1
Total Memory Required For Aerodynamic Functions and Breakpoints	1000	1675
Average Number of Breakpoints per Function	4.8	6.9

Table V I UDOFT and TX-O Data Storage for Aerodynamic Function Generation

It can be seen that the TX-O uses 67.5% more storage than UDOFT. However, the TX-O method uses 44% more breakpoints on the average than does the UDOFT method. If the number of breakpoints for all variables is increased by a certain percentage, the total function storage increases by more than that percentage. Therefore, a much more valid comparison is to compute the storage requirement for the TX-O method using the UDOFT average of 4.8 breakpoints per function. On this basis, the TX-O storage requirement is only 859 registers, actually 14.1% less than UDOFT required. This figure assumes only one breakpoint table per variable which probably would not be sufficient for this reduced number of breakpoints. However, even with more than one breakpoint table per variable, the TX-O method, scaled down to UDOFT accuracy, still compares favorably with the UDOFT method in memory requirement.

4.5 Comparison of Operating Times

Comparisons of actual running time for function generation between the TX-O and UDOFT are not particularly meaningful due to differences in order code and speed of operation between the two machines. Such a comparison does not afford a valid means of evaluating the two methods of generating functions. For this purpose, the time required by the TX-O to carry out the computations using the UDOFT method of function generation will be compared to the time required by the TX-O using the TX-O method of function generation. In the TX-O program, a function of one variable requires 109 μsec. For just the aerodynamic functions alone, the TX-O would take 8.720 msec. to calculate the 80 functions of one variable used by UDOFT. The TX-O takes 10.846 msec. to calculate the non-linear functions used for aerodynamic coefficients employing the TX-O method. Thus, the TX-O method is 24.4% slower in just calculating the functions themselves. The times given above do not constitute the total time for generating aerodynamic coefficients, however. For the UDOFT method, it is necessary to add to the above estimate the time required to multiply and add the functions of one variable to form the composite aerodynamic coefficients. The time given above is only for the calculation of each of the individual functions. The additional time required to combine functions would decrease the advantage of the UDOFT method over the TX-O method in computation time.

4.6 Results of Comparisons

The comparison of the two types of function generation reduces to a consideration of four factors. The first two are running time and memory storage requirement. The memory storage requirement is approximately similar when the accuracy of the TX-O method is reduced to the same level as that of the UDOFT method. The UDOFT running time seems to be slightly less than the TX-O running time, the difference probably being somewhere under 10% The other two considerations are slightly more abstract. The accuracy of the TX-O method has been scaled down to the UDOFT level for the storage requirement comparison. However, the running time is independent of accuracy. If the full number of breakpoints is used for the TX-O method, thereby increasing the storage requirement, the running time for the TX-O would still be only about 10% greater than the UDOFT method running time. The TX-O method accuracy in this case would be significantly better. The last factor is perhaps the most abstract, and yet for some applications the most important. The amount of preliminary data processing required and ease of modification of the existing program are clearly factors in which the TX-O method excels over the UDOFT method.

In conclusion, it would appear that for problems where the accuracy requirement is known in advance and is not stringent, and where preliminary analysis and ease of modification are definitely not important factors, the UDOFT scheme would probably be acceptable, especially if running time were a critical factor. In other problems where the accuracy might have to be increased or where there was a good chance of future modifications being required, the TX-O method would be preferred.

4.7 Modification to the Order Code for Function Generation

There are only two basic operations in function generation, level selecting and generating a function of one variable by linear interpolation between discrete points. Functions of two and three variables just involve repeated application of the single function generation procedure.

A convenient instruction for speeding up the level-selecting process is incorporated in the Bendix G-20 order code.[13] The command recommended for the TX-O which will be called level select, cxs x, consists of a repeated add and test sequence. First, the contents of the effective address are added to the accumulator. The effective address is formed by adding the instruction address, x, to the contents of the index register. If the contents of the accumu-

lator after the addition is positive, then the magnitude of the index register is decreased by one, the live register is placed in the accumulator, and the instruction is repeated. Of course, on the next cycle the effective address is different since the index register has changed. This process continues until the result in the accumulator is negative at which time the instruction following the cxs x instruction is performed. A block diagram of the operation of the command is shown in Figure 11.

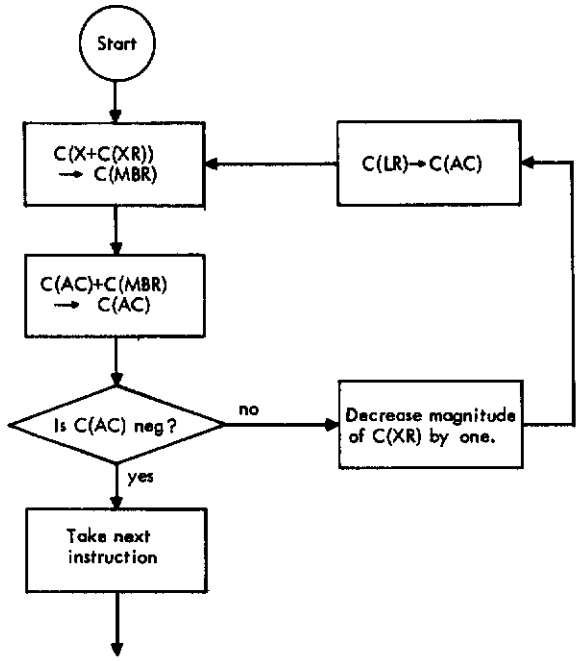


Fig. 11 Operation of the Level-Select Instruction, cxs x

The instruction would take 12 μsec. for the first operand, the 6 μsec. for each additional operand until the sequence terminates.

The following sequence of instructions performs the level-select operation using the zone from the previous solution so that the iteration goes through three trials at most. The notation used is that x is the variable on which the level selecting is performed, i is the zone number from the last solution cycle, and the zone number for the present solution cycle is k which will differ from i by at most one. The comment following each instruction refers to the contents of the registers affected by the instruction after the instruction has been performed.

Address tag	Instruction	Comment
beg,	ldx xir	c(xr) = i
	aux (1	c(xr) = i+1
	llr (x	c(lr) = x
	lcc	c(ac) = -x [lcc = cla + com + lmb + pad]
	alr	c(lr) = -x [alr = amb + mbl]
	cxs xtb	level-select command when the repeated sequence terminates: c(ac) = x_i - x and c(xr) = k
	iadUcom	c(lr) = x_i - x and c(ac) = - x_i [iad = amb + (mbl + lmb) + pad + cry]
	adx xtb+1	c(ac) = $x_{i+1} - x_i$
	ialUcom	c(lr) = $x_{i+1} - x_i$ and c(ac) = x - x_i [ial = amb + cla + (mbl + lmb) + pad]
	dvf *	c(ac) = $(x - x_i)/(x_{i+1} - x_i)$
	sto sx	c(sx) = $(x - x_i)/(x_{i+1} - x_i)$
end,	sxa xir	c(xir) = k

Address tag	Data	Comment
xtb,	x_0	
	x_1	
	x_2	x breakpoint table
	.	
	.	
	.	
	x_7	
sx,	-	storage for ratio $(x - x_i)/(x_{i+1} - x_i)$
xir,	i	x zone from previous solution cycle

* dvf = fraction divide, an assumed order discussed on p. 25

This sequence requires 190 μsec. in the worst case where the zone number has decreased by one from the previous cycle. Compared to the 340 μsec. that level select takes with the present TX-O order code and a 40 μsec. divide, the new command would save 150 μsec. for each level select or 1950 μsec. for the thirteen level selects in the worst case.

If additional breakpoint tables were required, they would add 190 μsec. each to the worst-case running time of the program.

The level select command defined here assumes that the variable lies within the allowed range. This could be checked elsewhere in the program when the variable is calculated.

The operation of generating a function of one variable after the level selecting is done appears to be too complicated to consider implementing by a single instruction. The time presently required for generating a function of one variable is 109 μsec. of which 24 μsec. represents instructions which would still be needed even if a single-instruction function generate were used. 25 μsec. of the remaining 85 μsec. is taken up by multiplication which will be discussed in a later chapter. Additional arithmetic instructions, such as a clear and add instruction and a subtract instruction, also decrease the amount of time required for this operation. These instructions will be discussed in the next chapter.

The conclusion is that the part of single function generation that would adapt to direct implementation by circuitry does not seem to be time-consuming enough to warrant the additional control circuitry necessary.

4.8 Summary of Conclusions for Function Generation

The relative advantages and disadvantages of the TX-O method and the UDOFT method of function generation are summarized below.

Advantages of the TX-O method over the UDOFT method:

1) The TX-O method is more accurate with the number of breakpoints used for the F-100A problem.
2) The TX-O method requires less preliminary data processing when the program is written.
3) The TX-O method can be modified more easily.
4) The accuracy of the TX-O method can be extended much more easily.

Advantages of the UDOFT method over the TX-O method:

1) The UDOFT method would be about 10% faster than the TX-O method on the same machine.

2) The UDOFT method requires less data storage for the number of breakpoints used. (If the accuracy of the TX-O method is scaled down to the accuracy of the UDOFT method, then the memory storage requirement is approximately the same.)

3) Each function has its own breakpoint table in the UDOFT method.

CHAPTER V

MODIFICATIONS OF THE TX-O ORDER CODE

5.1 Introduction

From the familiarity gained by completely programming the F-100A problem on the TX-O, the advantages of making certain modifications became apparent.

The four modifications discussed in this chapter all save a considerable amount of running time, and they would therefore be recommended for inclusion in the order code of a computer for real-time simulation. Other possible modifications that would save a lesser amount of time than the four discussed in this chapter are discussed in Chapter VIII on Inputs, Outputs, and Decisions. These other modifications cannot be definitely recommended since the amount of time they save does not seem to justify the complexity of implementing them.

5.2 Level-Select Order

One potential modification of the TX-O order code has already been discussed. It is the level-select order introduced in the last chapter.

cxs x level-select	Find the first operand c(z) which when added to the accumulator results in a negative accumulator. Decrease the magnitude of the index register by one after each unsuccessful iteration. z = x + c(xr)	execution time 6+6n μsec. n is the number of iterations

The use of this instruction saves 1.950 msec. if there are thirteen level selects and each represents a worst case. If additional breakpoint tables are used, each additional level select and slope generation would require 190 μsec. as compared to the 340 μsec. required for a level select without the special order. The savings in memory using the level select command would be 105 registers for the fifteen level selects in the entire program. Each additional

breakpoint table would require an additional twelve instruction sequence as compared to nineteen instructions without the level select order.

5.3 Multiply Order

The TX-O has a five bit instruction code. Of the thirty-two possibilities, the existing operate code structure eliminates eight, thereby leaving a maximum of twenty-four addressable commands. The future TX-O order code described in Appendix I includes eighteen addressable instructions, leaving only six unused addressable commands. The original reason for making the multiply command non-addressable was to conserve these few remaining addressable instructions. However, an examination of the TX-O program reveals that an addressable multiply command is practically imperative. With few exceptions, every multiply instruction in the present program is preceded by a load live register instruction. Both factors that are to be multiplied are practically never in the arithmetic unit (accumulator and live register) when the multiplication is to be performed. As a result, the multiply is effectively a two-instruction sequence requiring 37 μsec. rather than a single instruction requiring 25 μsec. The multiply command entry in Table II includes only the non-addressable multiply command itself. If the load live register instruction were included, then the worst case multiply operations would require 14.541 msec. or 27.0% of the worst case running time.

The multiply command suggested for inclusion in the TX-O would be:

mpy x multiply	Fraction multiply the contents of register x by the contents of the accumulator leaving the product in the accumulator	execution time 25 μsec.

The savings in worst case time using the addressable multiply would be the number of worst case multiplies times the operation time of a load live register command. For 393 worst-case multiplies this comes to 4.716 msec. The memory savings of 453 registers is equal to the total number of multiplies in the program.

5.4 Load Accumulator Order

There are two other commands that the TX-O does not presently have that would save an appreciable amount of time for this problem: The load accumulator, or clear and add instruction, described in this section, and the subtract instruction described in the next section. Both commands would be indexable. Neither command is particularly unusual and, in fact, most general purpose machines incorporate them in the basic order code.

lda x load accumulator	c(x) → c(ac)	execution time 12 μsec.
lax x load accumulator indexed	c(x+c(xr)) → c(ac)	execution time 12 μsec.

The operation of a load accumulator instruction is actually simpler than an add instruction since there is no carry. The accumulator would be cleared at the beginning of the operand cycle of the instruction and when the operand is available in the memory buffer register it would be partially added to the accumulator. Since the TX-O is not an asynchronous machine and the time gained by eliminating the carry could not be utilized, it might be simpler on the TX-O to have the load accumulator instruction initiate a full add. The only difference would be that for a load accumulator instruction the accumulator is cleared before the memory buffer register is added to it.

The savings in worst-case time using the unindexed load accumulator instructions is 2.352 msec. The memory saved is 230 registers. The savings due to the indexed load accumulator instruction cannot be specified independently of the savings due to the subtract instruction and will therefore be presented with the subtract instruction in the next section.

5.5 Subtract Instruction

Subtraction on the TX-O is presently accomplished by using an operate command to complement the accumulator in conjunction with a subsequent add command. This two-instruction sequence takes 24 μsec. The following

subtract commands would be useful for the aircraft simulation problem, particularly in the function generation routines.

sub x subtract	c(ac) - c(x) → c(ac)	execution time 12 μsec.
sux x subtract indexed	c(ac) - c(x +c(xr)) → c(ac)	execution time 12 μsec.

The subtract instruction can be implemented on the TX-O in any one of three ways. The first is to complement the memory buffer register before it is added to the accumulator. This is not a particularly good method, since the memory buffer register cannot be changed until the operation of reading the word back into memory is completed. This would require waiting until very late in the cycle to complement and add. A better way might be to complement the accumulator before the memory buffer register is added and once again after it is added. This would not be difficult since the circuitry to complement the accumulator is already there. The third method would be to modify the add circuit to subtract also. For subtraction the partial add proceeds as with addition. The subtract carry (called borrow for subtraction) differs slightly in functional form from the add carry but is propagated in the same way.

The subtract indexed is used with the load accumulator indexed instruction to save time in the function generation routines. The worst-case time saved by both of these instructions is 2.172 msec. The savings in memory is 215 registers.

A summary of running time and memory saved by all these instructions is shown in Table VII.

Incorporating all the modifications proposed in this chapter would result in a reduction in the total worst-case running time and memory requirements stated in Chapter III by the totals shown in Table VII. The new worst-case running time would be 42.725 msec. and the program would require a total of 6913 registers.

Instruction	Worst-Case Time Saved (msec.)	Memory Saved (registers)
level select	1.950	105
addressable multiply	4.716	453
load accumulator	2.352	230
load accumulator indexed subtract subtract indexed	2.172	215
Total	11.190 msec.	1003

Table V II Time and Memory Saved Using Additional Instructions

CHAPTER VI

THE USE OF SUBROUTINES IN REAL-TIME FLIGHT SIMULATION

6.1 Introduction

All of the modifications discussed so far have decreased both running time and memory requirements. If, for various reasons, one wishes to decrease running time, it can be done at the expense of additional memory, or, conversely, if one wishes to decrease the amount of memory required, it can be done at the expense of increasing the running time. An excellent example of this type of trade-off between running time and memory arises in the use of subroutines.

6.2 Subroutines for Function Generation

For the F-100A simulation problem there are only three places where subroutines could possibly be used to any substantial advantage. The first of these is function generation.

Since running time was considered the most difficult of the specifications to meet, no subroutines were used in the initial TX-O program. This resulted in a fast-running program which was relatively inefficient in memory utilization. If the problem can be solved in less than the desired solution cycle time, then it is possible to use some of this surplus time to decrease the amount of memory required.

As an example of the use of subroutines to decrease memory requirements at the expense of time, consider the function generation problem. In TableVIII the breakdown of non-linear functions is shown. The first column is the number of functions of a particular variable encountered in the worst-case running time. The second column is the number of functions not encountered in the worst-case running time. These latter functions do not effect worst-case running time, but they do influence the memory requirements. The third column is the sum of the first two columns.

Two types of subroutines were investigated for function generation. The first type used one subroutine for all functions of one variable and one subroutine for all functions of two variables. The second type used one subroutine

	Number in Worst-Case Timing	Number Skipped in Worst-Case Timing	Total Number
Functions of 1 Variable			
h	15	19	34
M_a	17	2	19
V_T	3	4	7
α	5	0	5
$\dot{\gamma}$	3	1	4
q	2	0	2
RPM	0	2	2
N_2	1	0	1
β	1	0	1
δR	0	1	1
% Fn	0	1	1
M_f	1	0	1
Total	48	30	78
Functions of 2 Variables			
q and Ma	13	0	13
α and Ma	8	0	8
h and Ma	6	0	6
C_L and Ma	2	0	2
δ J and Ma	1	0	1
α and δA	1	0	1
δ A and Ma	0	1	1
Total	31	1	32
Functions of 3 Variables			
α, Ma, and δA	1	0	1

TableVIII Breakdown of Non-Linear Functions for TX-O Program

for each variable for functions of one variable and one subroutine for each pair of variables for functions of two variables. The advantage of the second type over the first is that the calling sequence is shorter for the second type since the information that applies to all functions of a particular variable or pair of variables does not have to be included in the calling sequence. The disadvantage of the second type is that each additional variable or pair of variables requires an additional subroutine. It is clear that for variables used only for one function, such as β, there is no advantage to writing a separate subroutine to handle a single function. In fact, due to the length of the subroutine, the break-even point for functions of one variable is between two and three functions. Therefore, in analyzing the second type of function generation, only variables with more than two functions will utilize subroutines. For functions of two variables, the break-even point is between one and two functions so that the second type of subroutine will be used for all pairs of variables used in two or more functions.

In analyzing the first type of subroutine, it does not matter how few functions there are of a particular variable, and therefore all functions of one and two variables will be computed by subroutine.

In view of the fact that the additional arithmetic instructions recommended in the last chapter have a great effect on the amount of time and memory required by function generation, the use of subroutines was examined both with and without these additional instructions. Table IX presents the results without the use of the addressable multiply, load accumulator, and subtract instructions and Table X presents the results obtained with the use of those additional orders.

It can be seen that in every case the use of the second type of subroutine, with a subroutine for each variable, is to be preferred. This statement is true for this particular application of subroutines, but it cannot be considered a general result. For a problem with more variables and fewer functions of each variable the method of using one subroutine for all functions of all variables would probably yield better results.

The reduction of memory in Table IX is somewhat hypothetical for the particular simulation problem at hand. Without the use of the instructions recommended in Chapter V, the program runs over fifty milliseconds in the worst case. Therefore, no savings in memory can be effected because there is no surplus time available to trade off for that memory. Again it should be emphasized that subroutines cannot be used to reduce memory requirements

	Total Memory Requirement	Memory Saved Over Corresponding No-Subroutine Case	Total Worst-Case Running Time (msec.)	Extra Running Time Over Corresponding No-Subroutine Case (msec.)	Memory Saved Divided By Additional Running Time In msec.
Functions of 1 Variable					
No Subroutines	624	-	5.232	-	-
1 Subroutine For All Functions	476	148	8.400	3.168	46.7
1 Subroutine For All Functions of Each Variable	339	285	8.070	2.838	100.4
Functions of 2 Variables					
No Subroutines	672	-	9.021	-	-
1 Subroutine For All Functions	249	423	12.555	3.524	120.0
1 Subroutine For All Functions of Each Pair of Variables.	250	422	10.935	1.914	220.5

Table IX Results of Using Subroutines With Present TX-O Order Code

	Total Memory Requirement	Memory Saved Over Corresponding No-Subroutine Case	Total Worst Case Running Time (msec.)	Extra Running Time Over Corresponding No-Subroutine Case (msec.)	Memory Saved Divided By Additional Running Time In msec.
Functions of 1 Variable					
No Subroutine	468	-	4.080	-	-
1 Subroutine For All Functions	398	70	7.824	3.744	18.7
1 Subroutine For All Functions of Each Variable	315	153	6.402	2.322	54.9
Functions of 2 Variables					
No Subroutine	448	-	6.417	-	-
1 Subroutine For All Functions	211	237	9.941	3.524	67.3
1 Subroutine For All Functions of Each Pair of Variables	197	251	7.983	1.566	160.3

Table X Results of Using Subroutines With Additional Arithmetic Instructions

unless the problem is solved with time to spare. If some means of reducing the running time were found, other than those mentioned in Chapter V, then it would be possible by using subroutines for function generation to free 422 registers of memory by increasing the worst case running time 1.914 msec. or to free 707 registers by increasing the worst case running time 4.752 msec.

The program with the instructions recommended in Chapter V does have extra time that could be traded for memory. With the addressable multiply, the load accumulator and the subtract commands, the worst case time would be 44.675 msec. and the memory required would be 7018 registers. This leaves 5.325 msec. of worst case time available. From Table X it can be seen that 251 additional registers can be freed at a cost of 1.566 msec. or 404 registers can be freed at a cost of 3.888 msec. Using this last strategy, the alternate worst case running time and memory requirement figures would be 48.563 msec. and 6614 registers respectively. Intermediate values of memory and time can be obtained by doing fewer of the functions by subroutine.

6.3 Subroutines for Other Parts of the Program

The other parts of the program where subroutines could be used to advantage are level selecting and integration. The memory saved using subroutines in either of these parts is less than 100 registers, hence they will not be examined in detail here. The ratio of the memory saved to additional running time shown in Tables IX and X is a good means of comparing the value of subroutines in any particular application.

6.4 Conclusions

If the modifications suggested in Chapter V are incorporated into the TX-O or if the worst case operating time is reduced significantly below 50 msec., then some or all of this extra time may be used to reduce the amount of memory required. The most memory capacity can be saved by incorporating subroutines into the function generation routine. If still more time is available, then a modest amount of memory can be gained by using subroutines for level selecting and integration.

CHAPTER VII

INTEGRATION AND WORD LENGTH

7.1 Introduction

The first part of this chapter discusses the integration methods used by UDOFT and the TX-O programs. The second part of this chapter discusses word length requirements and the way in which integration affects word length.

7.2 Integration

UDOFT used a method of integration developed at the University of Pennsylvania called Mod Gurk.[10] Mod Gurk uses the past three values of the functions and the past three values of the derivative:

$$x(n) = a\,x(n-1) + b\,x(n-2) + c\,x(n-3) + h\left[d\,\dot{x}(n-1) + e\,\dot{x}(n-2) + f\,\dot{x}(n-3)\right] \qquad (7.1)$$

where:

$a = 1.1462084$
$b = -0.2010870$
$c = 0.0548788$
$d = 1.6415880$
$e = -1.0080120$
$f = 0.2750960$

The discrete time index is n and h is the solution cycle time, which for the F-100A must be not greater than 0.05 seconds.

In order to evaluate the accuracy of the Mod Gurk integration formula and several other more common integration methods, the M.I.T. Electronic Systems Laboratory conducted an experimental study of integration methods on the Whirlwind computer.[1] The equations of motion for the F-100A were solved on a non-real-time basis for several integration methods and solution rates. One of the results of this study is reproduced in Figure 12.

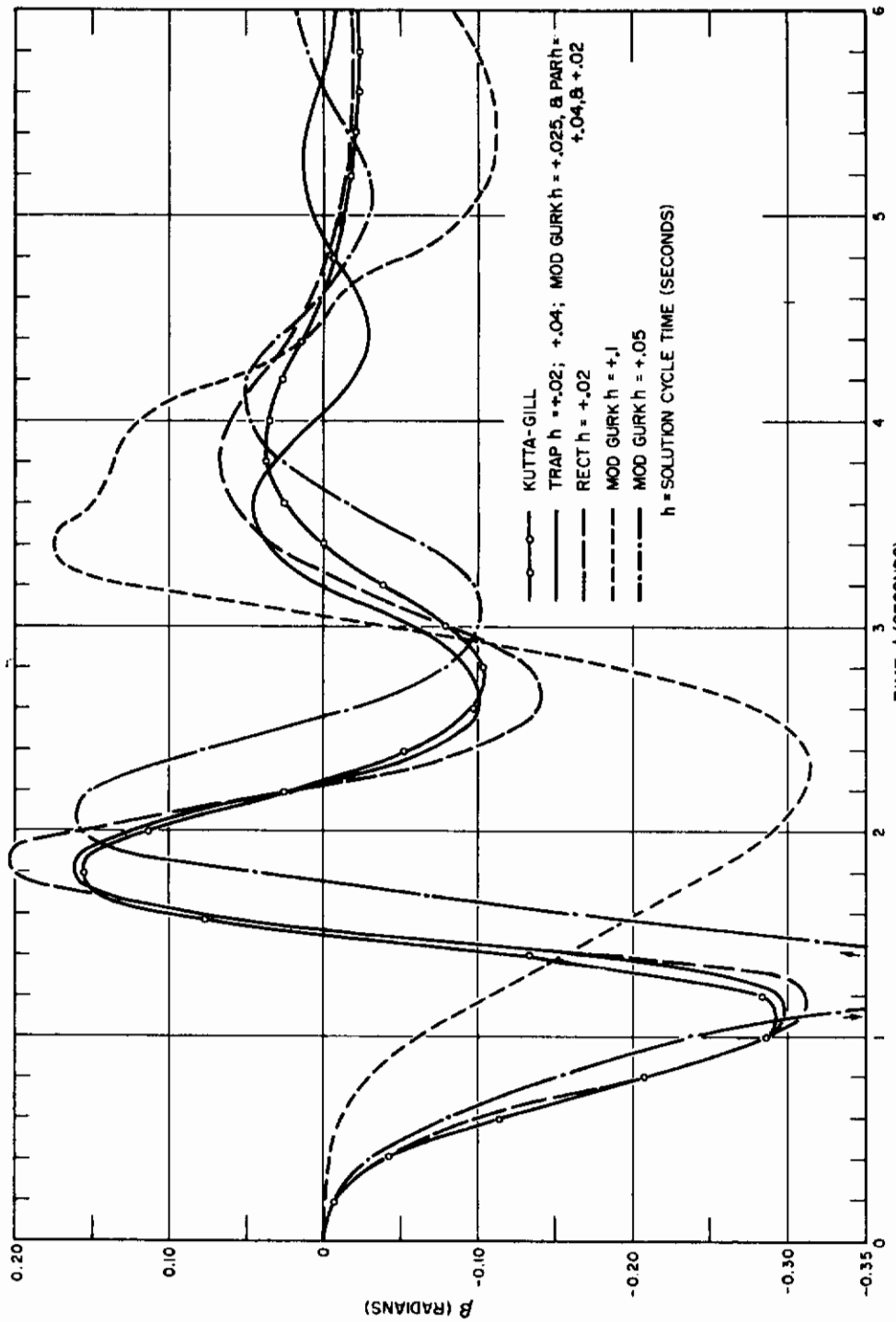


Fig. 12 Effect of Solution Rate and Integration Formula on a Simulated Aircraft Transient

The other integration methods used are:

Rectangular

$$x(n) = x(n-1) + h\,\dot{x}(n-1) \tag{7.2}$$

Trapezoidal

$$x(n) = x(n-1) + \frac{h}{2}\left[3\dot{x}(n-1) - \dot{x}(n-2)\right] \tag{7.3}$$

Parabolic

$$x(n) = x(n-1) + \frac{h}{12}\left[23\dot{x}(n-1) - 16\dot{x}(n-2) + 5\dot{x}(n-3)\right] \tag{7.4}$$

The Kutta-Gill method is a complicated, closed-type formula that is much too time-consuming to be considered for real-time simulation. It is included only for comparison, since, for the purpose of this investigation, it can be considered as representative of the solution to the problem.

Figure 12 shows the response of β, the sideslip angle, to a moderate rudder pulse. Notice that Mod Gurk with h = 0.05 seconds appears to be less satisfactory than the other methods used with the exception of Mod Gurk with h = .1 sec. and rectangular with h = 0.02 seconds. In particular, trapezoidal integration with h = 0.04 sec. appears to be a good compromise between accuracy and computation time.

On the basis of this study the method selected for the TX-O program is the trapezoidal rule repeated here for convenience.

$$x(n) = x(n-1) + \frac{h}{2}\left[3\dot{x}(n-1) - \dot{x}(n-2)\right] \tag{7.5}$$

Notice that the only multiplication required is the multiplication by h/2 since $3\,\dot{x}_{n-1}$ can be computed faster by adding than by multiplication. If the solution cycle h were an integral power of two, then no multiplication would be required since the multiplication by h/2 could be accomplished by shifting.

In the TX-O program there are only twelve integrations; six accelerations, and six direction cosines. Each integration requires 133 μsec. With an addressable multiply and a subtract instruction each integration would require 97 μsec.

7.3 Word Length - Introduction

There are two ways of arriving at the minimum word length required for a particular problem. The first, to be discussed in the next section, depends on the number of instructions and the size of directly-addressable memory required. The second method, which will be discussed in the remaining sections, depends in a rather complicated way on integration, scaling, the pilot's ability to introduce small control changes, and the pilot's ability to perceive small changes on the cockpit instruments.

7.4 Instructions, Addressable Memory, and Word Length

The TX-O presently has eighteen addressable instructions listed in Appendix I. Of these eighteen, one instruction, the add one to memory (ado x) instruction, is definitely not used in the TX-O simulation program. Three other instructions are useful only if subroutines are incorporated. These three instructions are: store index in address (sxa x), transfer and set index (tsx x), and unconditional indexed transfer (trx x).

The use of five bits for addressable instructions and the use of the present operate class structure leaves six unused addressable instructions on the TX-O. Without the add one instruction there would be seven, and without the three instructions useful for subroutines, there would be ten unused instruction codes. There are six addressable instructions definitely recommended in Chapter V. Incorporating these six would still leave one to four positions unused if the instructions deemed not useful were deleted. From the preceding discussion it would seem that five bits of instruction code are sufficient for the F-100A problem even with the present operate class structure, which decreases the maximum number of addressable instructions 32 to 24.

If additional instructions are needed there are two possibilities. One is to add additional bits to the instruction code. The other is to decrease the number of bits used by the operate class commands so that there are 32 usable addressable instructions or actually 31 addressable instructions with the 32nd being used to indicate an operate class instruction. This last possibility would reduce the versatility of the operate class structure unless the total word length were increased proportionately. The usefulness of the operate class commands on the TX-O is unquestionable. However, with the addition of extra instructions such as load accumulator, subtract, and addressable multiply, the number of operate class instructions used decreases con-

siderably. It is possible that with the inclusion of these additional addressable instructions, the complexity of the operate class structure could be decreased without losing any program efficiency. It is also possible that if the complexity of the operate class structure were decreased, operate class instructions would require only one cycle time (6 μsec.) rather than the present two cycle times (12 μsec.). This would make the use of the live register as temporary storage more advantageous than at present.

The second factor in the word length requirement imposed by instructions and memory is the amount of directly addressable memory required. It is quite evident that 8192 words of memory are sufficient for the F-100A problem, since both the UDOFT and the TX-O programs use somewhat less than this amount. One approach to the problem is to say that 8192 words require thirteen bits of address section. This may be more bits than are needed, however. There are two methods of utilizing larger memories than can be directly addressed.

The first method is illustrated by the Digital Equipment Corporation's PDP-1.[14] This computer has a twelve bit memory address section which is sufficient to address 4096-word banks of memory; the programmer must specify which bank of memory the instructions following will refer to. This is done with a jump field instruction (jfd y), where y is the bank number selected. All instructions refer to this bank until the next jump field instruction. This method is not as restrictive to programming as it might at first appear to be. If the entire program is written with a knowledge of this addressing method, then very little time need be lost due to changing fields. This is especially true in the TX-O program which uses very few subroutines. Handling an 8192 word memory by this method will only save one bit of memory address section, so for the F-100A problem this method does not seem to be very advantageous.

The other method is that used by Control Data Corporation's 160 and 160A computers.[15] These computers have only a six bit memory address section, but the six bits can be interpreted in any one of five ways: 1) The six bits can specify one of the first 64 words of memory. 2) The six bits can specify one of the first 64 words of memory which contains the address of the register to be used. 3) The six bits can specify one of the 64 words of memory directly following the instruction being performed. 4) The six bits can specify one of the 64 words of memory directly preceding the instruction being performed. Or 5) The six bits can be used as the operand itself.

This method is extremely efficient in word length; but for problems such as the F-100A, with large tables of data, it could not be used.

Of course, there is no reason why either of these methods could not be used with memory address sections of other than 4096 or 64 words. For some problems, some other memory address section size might be better.

The conclusion arrived at in this section is that for the F-100A problem eighteen bits is just barely sufficient for instructions and memory. This is the actual word length requirement for the problem only if the other factors effecting word length, such as integration, require a shorter word than this.

7.5 Effect of Integration on Word Length

Before discussing the effect of integration on word length, it is necessary to understand the scaling method used.

A possible method of scaling a variable on a binary computer consists of setting the maximum value of a variable equal to the largest binary number the register will hold. For a variable x with maximum value $x_{max.}$ this procedure is illustrated below for an n-bit word including sign bit:

$+x_{max}$ is represented by

s	1	2				n-2	n-1
0	1	1	•	•	•	1	1

This procedure is very difficult to handle since the scaling coefficient k defined by the variable x and its machine representation X,

$$x = k \cdot X \tag{7.6}$$

is not usually an integral power of two and therefore multiplication is necessary before X can be added to another variable with a different scale factor.

Another method of scaling that is not quite as efficient in word length utilization, but is much easier to handle, is the one used for the TX-O program. It involves using the actual binary number in the register to represent the variable with a power of two as the scale factor. The relation between the variable x and its machine representation X in fraction form is now

$$x = 2^{s} \cdot X \qquad \text{where } 1 > |X| \geq 0 \tag{7.7}$$

where s is a positive or negative integer called the scale factor of the machine representation X. The scale factor, s, is chosen so that $+x_{max}$

is represented by a one in the most significant bit of X. The largest number that can be reproduced in a register n bits long including the sign and with scale factor s is

$$2^s - 2^{s-n+1} = 2^s(1-2^{-n+1}) \approx 2^s \text{ if } n >> 1 . \tag{7.8}$$

The smallest increment that can be represented in the same register is

$$2^{s-n+1} . \tag{7.9}$$

A procedure for finding the optimum scale factor, s, is to find the solution of the following inequality:

$$2^s(1-2^{-n+1}) \geq x_{max} > 2^{s-1}(1-2^{-n+1}) . \tag{7.10}$$

If n >> 1 the scale factor usually satisfies the simpler inequality:

$$2^s > x_{max} \geq 2^{s-1} . \tag{7.11}$$

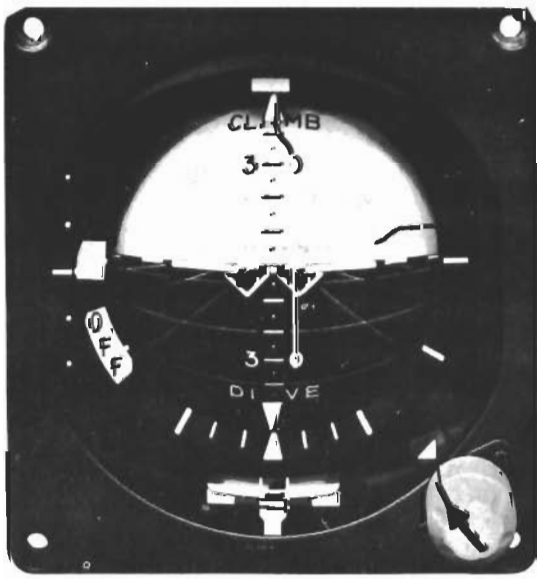
We are now prepared to discuss the effect of integration on word length. The integration requirement imposed by altitude is, in the case of the F-100A, the most demanding in word length. To see how this comes about, it is necessary to understand a few of the pilot's instruments and to make an assumption concerning the pilot's ability to read the instruments.

The altimeter, shown in Figure 13 a, has three pointers: hundreds of feet, thousands of feet, and tens of thousands of feet. The most sensitive scale is 1000 feet for a full revolution. If the assumption is made that the pilot can detect a movement equal to one part in a thousand of the full scale deflection, then, the pilot would be able to detect a pointer movement of 0.36° or an altitude change of one foot. Thus, the minimum increment of altitude that must be simulated due to the pilot's ability to read the altimeter is one foot.

The vertical speed indicator, shown in Figure 13 b, is a much more sensitive instrument. Full scale on this meter is ±2000 feet per minute. Using 0.36° as the minimum change the pilot can detect and noticing that the scale is non-linear, the minimum increment that the pilot can detect around zero vertical speed is 0.046 feet per second. We will call this value $\dot{h}_{min}$. The integration formula to be used for this example is the simple rectangular rule:

a) AIRCRAFT ALTIMETER

b) VERTICAL SPEED INDICATOR



c) GYRO HORIZON

Fig. 13 Aircraft Instruments

$$h(i) = h(i-1) + \dot{h}(i-1)\,\Delta t \tag{7.12}$$

where i is the discrete time index and Δt is the solution cycle time, which for the F-100A program is 0.05 seconds.

The important factor affecting word length is that the pilot can detect a vertical speed of $\dot{h}_{min}$ on the vertical speed meter, and he then expects the altimeter to change. The altimeter change will admittedly be slow for such a small vertical speed, but if the pilot maintains a small non-zero vertical speed then he anticipates that the altimeter will change at a corresponding rate. Thus, the word length of the register representing altitude, h, must be long enough so that $\dot{h}_{min}$ multiplied by Δt and shifted to take into account differences in scaling between h and $\dot{h}$, will add a non-zero increment to h on each cycle of the integration. This means that $\dot{h}_{min}$ times Δt must be greater than or equal to the smallest increment in altitude that can be represented in the n bit register:

$$\dot{h}_{min} \cdot \Delta t \geq 2^{s-n+1} \tag{7.13}$$

where s is the scale factor for altitude. In order to demonstrate this procedure, the actual numbers for the F-100A will be used. First the scale factor of h must be found. This is done using equation 7.11. The maximum altitude for the F-100A is 55,500 feet.

$$2^s > 55,500 \geq 2^{s-1} \ . \tag{7.14}$$

The solution to this inequality is s = 16 which is the scale factor for the altitude register. Using this value in equation 7.13 we see that the word length n must satisfy the inequality:

$$\dot{h}_{min} \cdot \Delta t = (0.046)(0.05) = 0.0023 \geq 2^{16-n+1} = 2^{17-n} \ . \tag{7.15}$$

Of course the smallest n which satisfies this inequality is the one desired and this is n = 26. Therefore, under the assumptions made, altitude requires a 26 bit word length including sign.

Certain interesting characteristics of the word length can be seen from equation 7.13. First of all n increases as Δt decreases. In other words, the word length requirement increases with the solution rate. Therefore, the minimum solution rate that is acceptable due to accuracy and stability

requirements will result in the shortest word length required for integration. Also notice that when h_{max} increases, n increases due to an increase in s. This is obvious, since if the maximum value of a variable increases, then the dynamic range must increase also; and therefore, the variable will require a longer word length.

If the solution cycle, Δt, is an integral power of two, then a graphical method suggested by Connelly[3] shows the relationship between $\dot{h}_{min}$ and the altitude word length quite clearly. As an example, take $\Delta t = 2^{-5} = 1/32$ second. The word diagram in Figure 14 shows that a 27 bit word length including the sign bit is the smallest that will satisfy the requirement that $\dot{h}_{min} \cdot \Delta t$ add a non-zero increment into h on every cycle. The scale factor of $\dot{h}$ has been assumed to be s = 11 as it is in the UDOFT program.

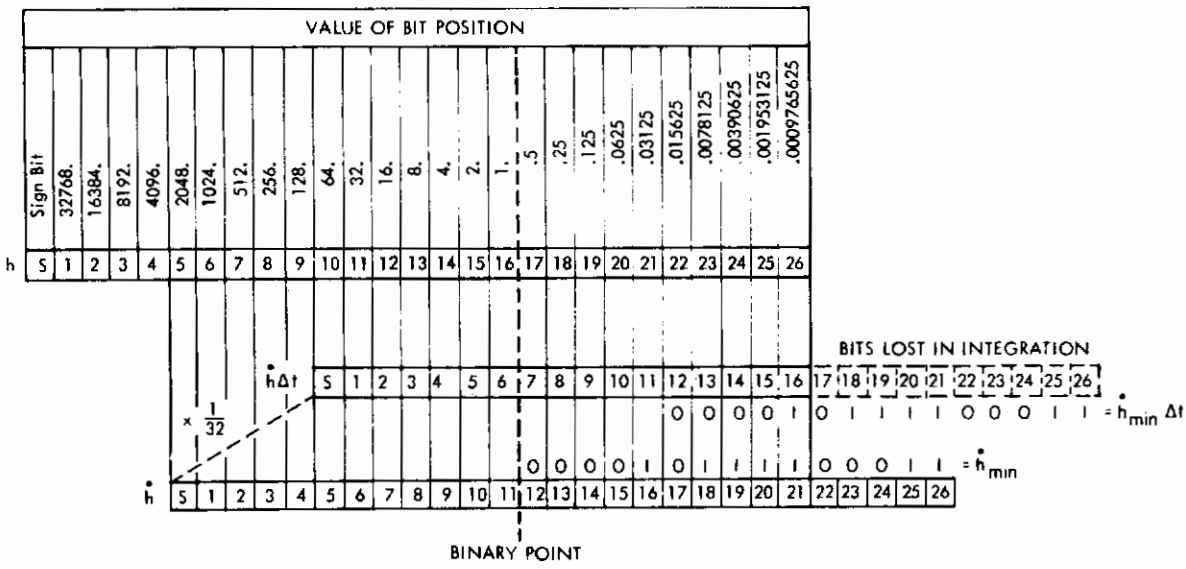


Fig. 14 Word Length Diagram for Altitude

Notice that increasing the solution rate from 20 to 32 solutions per second increased the word length requirement by one bit. The word diagram points out a property that was not obvious from the above discussion. In Figure 14, bit 16 of $\dot{h}$ would be the most significant bit of the representation of $\dot{h}_{min}$. However, bit 16 would not be the only bit equal to one. Therefore, since only bit 16 is added into h every cycle, there is an error in altitude caused by the other bits of $\dot{h}_{min}$ that were shifted off the right hand end of the accumulator. Bit 16 of $\dot{h}$ represents a vertical speed of 0.03125 feet

per second rather than the 0.046 feet per second we wished to add into h on each cycle. This error, although large percentage-wise, is really unimportant since the pilot could not distinguish between the change in altitude caused by a vertical speed of 0.03125 feet per second and the change in altitude caused by a vertical speed of 0.046 feet per second.

There is another type of integration that has an even more subtle effect on the word length. This is the double integration to generate the Euler angles from the rotational accelerations. In the UDOFT and the TX-O programs, the Euler angles themselves are never computed. Both these programs use direction cosines, which were shown in Chapter II to be trigonometric functions of the Euler angles. There is an integration involved in going from the angular velocities to the direction cosines. Even though Euler angles are not used, double integration is still effectively being performed. The computation of roll will be used as an example. Rectangular integration and Euler angles will be used in the example for simplicity. The integration equations used are:

$$p(i) = p(i-1) + p(i-1)\ \Delta t \tag{7.16}$$

$$\phi(i) = \phi(i-1) + p(i-1)\ \Delta t \tag{7.17}$$

Where i is again the discrete time index and Δt is the solution cycle time. The equation used to compute $\dot{p}$ will be a simplified equation with only the forcing term due to aileron deflection and the damping term due to roll rate included:

$$\dot{p} = \frac{\rho V_T{}^2 Sb}{2 I_x} \cdot C_{\ell_{\delta A}} \cdot \delta A + \frac{\rho V_T Sb^2}{4 I_x} \cdot C_{\ell_p} \cdot p\ . \tag{7.18}$$

The term $C_{\ell_{\delta A}} \cdot \delta A$ is always positive and the term $C_{\ell_p} \cdot p$ is always negative so that the equation is in the form:

$\dot{p} = B \cdot \delta A - A \cdot p$ where A and B are positive.

For steady state flight the response of the aircraft to a step input in δA is:

$$p = \frac{B \cdot \delta A}{A} (1 - e^{-At}) \tag{7.19}$$

$$\dot{p} = B \cdot \delta A\ e^{-At} \tag{7.20}$$

where:

$$A = - \frac{\rho V_T Sb^2}{4 I_x} C_{\ell_p}$$

$$B = \frac{\rho V_T^2 Sb}{2 I_x} C_{\ell_{\delta A}}$$

At a constant altitude and airspeed, p is building up exponentially toward the constant value $B \cdot \delta A/A$. $\dot{p}$ is decreasing from $B \cdot \delta A$ to zero by subtracting a quantity equal to $A \cdot p \cdot \Delta t$ on each cycle. We have seen from Figure 14 that when an integration is done, certain bits and the right hand end of the quantity integrated do not affect the integration. What would happen if $\dot{p}$ became too small to affect p on a particular cycle. First of all p would not change on that cycle and, therefore, the next value of $\dot{p}$ computed by equation 7.18 would be the same as the last value. In other words, as soon as $\dot{p}$ becomes too small to add into p on a particular cycle, the value of p is frozen at its present value. There are two possibilities; either this value of p is close enough to $B \cdot \delta A/A$ to yield an acceptable representation of the transient or the final value of p is not acceptable.

A non-real-time program was written for the TX-O to simulate the affect of different word lengths on the final value of p for a small aileron input on the F-100A. The altitude used was 35,000 feet and the speed was Mach 1.4. The aileron input, δA, was chosen to give steady-state value of p equal to $1/3^\circ$ per second. For ease of computation the solution rate used was 32 solutions per second. The results of this program are shown in Table XI. If 90% of true value is arbitrarily chosen as the accuracy requirement for this minimum input in δA, then eighteen bits including sign is required.

It may seem inconsistent to require that 90% of the true value of p be reached in response to a minimum input, when, in discussing altitude, it was stated that the accurate reproduction of small perturbations was unimportant. The reason is that at a roll rate of $1/3^\circ$ per second the pilot would be able to detect a change on the gyro-horizon, shown in Figure 13c, in about one second. In altitude, the minimum increment that the pilot could read on the vertical speed indicator is 0.046 feet per second. This rate of climb or descent would require 21.8 seconds to develop a change of one foot on the altimeter. The pilot could not have any feeling for how much change

he should see on the altimeter in 21.8 seconds, although he does know that it should change; but he would have a feeling for how much change he should see on the gyro-horizon after only one second.

Word Length Including Sign Bit	Percentage of True Value Reached
15	0
16	56
17	75
18	93.6
19	93.6
20	98.4
21	99.0
22	99.6
23	99.9
24	99.9
25	100.01
26	99.99

Table XI Word Length Required By Roll

On the F-100A, the word length required by roll is longer than that required by the other Euler angles; but it is still less than that required by altitude. In the next section special methods of handling those variables that require long word length will be discussed.

7.6 Double Precision Methods

It is always possible to design a digital computer with as long a word length as is required by any variable. This method is uneconomical however, if only a few variables require a much longer word length than any others. The cost of the machine increases almost proportionately to the increase in word length. Therefore, it is profitable to examine some methods of handling these variables, such as altitude, on a computer with insufficient word length to permit straightforward programming.

One method requires hardware implementation. It is a form of double precision arithmetic which is built into the Bendix G-20.[13] On this machine all arithmetic registers are double length. There is a tag bit in every instruction which controls whether the instruction is performed single precision or double precision. If the tag bit specifies double precision, then the instruction refers automatically to two memory registers, the one addressed directly by the instruction and the memory register immediately following. Thus the only additional time required by a double precision computation is the extra memory access involved in performing each instruction. This is an extremely small penalty compared to normal double precision programs.

Another possibility is to actually program altitude and other variables that require special handling with double precision methods. This is the method used on the TX-O for altitude and fuel flow.

7.7 Method of Selecting the Word Length

The method of setting the word length for a computer to be built for a particular problem would involve making up a table of the maximum word lengths required by each of the significant variables. This table should also include the maximum word length determined from the number of instructions and amount of directly-addressable memory required. After the table is complete it should be rearranged in order of decreasing word length requirement. In this form, it is clear how much smaller a word length would be required if certain variables were handled by special double-precision methods. It would be advantageous to handle all variables by special methods until either the word length decrease for handling an additional variable were too small to justify the extra time or until the word length dictated by the number of instructions and amount of memory were encountered. If the latter were the case, then the methods discussed in Section 7.4 of decreasing the length of the memory address section of the instructions could possibly be used.

For computers such as the TX-O, where the word length is fixed, it is still important to know the word length requirements of certain key variables, since these variables must be handled by special programming.

CHAPTER VIII

DECISIONS, INPUTS AND OUTPUTS

8.1 Introduction

Discrete and analog inputs and outputs account for 7.6% of the worst-case running time. Decisions, excluding discrete inputs, account for 8.2% of the worst-case running time. These areas provide perhaps the widest choice for the computer designer as far as different instructions and different methods of implementation are concerned. Decisions are discussed in the next section and inputs and outputs are discussed in the following sections.

8.2 Decisions

The decisions used in the TX-O program can be broken down into nine types. Table XII gives the classification of decisions, the usual coding, and the running time for each branch of the decision. Table XIII gives the number of each type in the worst-case program, the total number of each type, and the total worst-case time consumed for each type.

It can be seen from Table XII that the load accumulator and the subtract instructions discussed in Chapter V would reduce the worst-case running time for decisions.

Another possibility for reducing the running time is the use of a few standard skip-type instructions. Three such instructions that are also part of the G-20 order code are defined below:

son x skip on negative storage	Skip the next instruction if c(x) is negative.	execution time 12 μsec.
soz x skip on zero storage	Skip the next instruction if c(x) is zero.	execution time 12 μsec.
sag x skip on accumulator greater than storage	Skip the next instruction if c(ac) > c(x)	execution time 12 μsec.

1. $c(x) \leqslant 0$

```
cla
add x
trn ⟶ 30 μsec.
 |      if c(x) ⩽ 0
 ↓
36 μsec.
if c(x) > 0
```

2. $c(x) \leqslant$ constant

```
cla
add x
add (-constant
trn ⟶ 42 μsec.
 |
 ↓
48 μsec.
```

3. $c(ac) \leqslant$ constant

```
add (-constant
trn ⟶ 18 μsec.
 |
 ↓
24 μsec.
```

4. $|c(ac)| \geqslant$ constant

```
trn .+2
com
add (constant
trn ⟶ 42 μsec.
 |
 ↓
48 μsec.
```

5. $c(ac) \geqslant c(x)$

```
llr x
lcd
trn ⟶ 30 μsec.
 |
 ↓
36 μsec.
```

6. $c(x) = 0$

```
cla
add x
trz ⟶ 30 μsec.
 |
 ↓
36 μsec.
```

7. $c(x) \geqslant c(y)$

```
llr x
lcc
add y
trn ⟶ 42 μsec.
 |
 ↓
48 μsec.
```

8. $c(x) \geqslant$ constant

```
cla
add x
trn .+2
com
add (constant
trn ⟶ 66 μsec.
 |
 ↓
72 μsec.
```

9. $c(ac) \leqslant 0$

```
trn ⟶ 6 μsec.
 |
 ↓
12 μsec.
```

add (constant means add A where A is the address where the constant is stored.

trn .+2 means transfer to the second instruction after the trn instruction if the contents of the accumulator is negative.

Table XII Classification and Coding of Decisions

	Number In Worst Case	Total Number	Time Consumed In Worst Case (μsec.)
1. $c(x) \leqslant 0$	45	51	1524
2. $c(x) \leqslant$ constant	23	29	1038
3. $c(ac) \leqslant$ constant	26	32	612
4. $\lvert c(ac) \rvert \geqslant$ constant	12	12	228
5. $c(ac) \geqslant c(x)$	7	8	210
6. $c(x)=0$	5	6	156
7. $c(x) \geqslant c(y)$	3	4	138
8. $\lvert c(x) \rvert \geqslant$ constant	2	3	132
9. $c(ac) \leqslant 0$	7	7	84
Total	130	152	4422

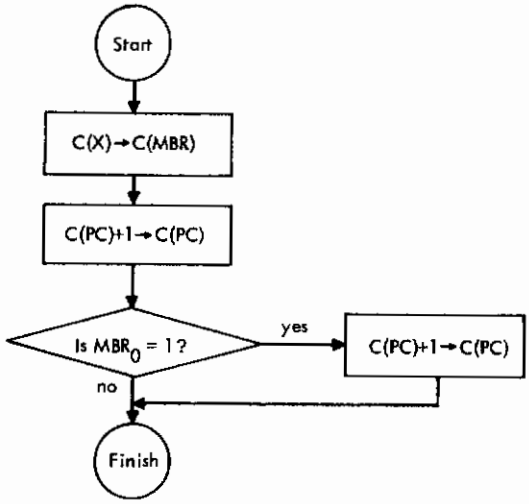
Table XIII Breakdown of Decisions by Types

The coding of the nine types of decisions using the skip instructions is shown in Table XIV. Notice that each skip instruction is followed by an unconditional transfer instruction. This decreases the time saved by the use of the skip-type instructions. Table XV shows the worst-case time saved using only the skip instructions, only the load accumulator and subtract instructions, and both types of instructions together. It can be seen that using all three skip-type instructions and both the load accumulator and subtract instructions saves only 747 μsec. over the time using just the load accumulator and subtract instructions alone. Using the skip-type instructions alone saves 1.431 msec. over the initial coding shown in Table XII The three skip instructions are not equally valuable. TableXVI gives the breakdown by instructions of the 1.431 msec. saved. None of the three instructions alone save as much as one millisecond, and for this reason, none of these instructions is considered useful enough to warrant the cost of installation on the TX-O for the F-100A problem. These instructions are also all addressable and require that the instruction code be expanded in order to include them.

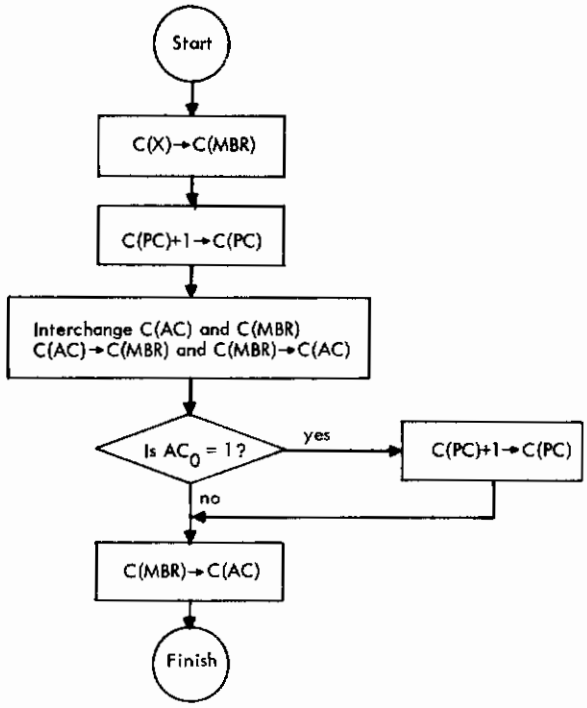
For a problem with many more decisions, these skip-type instructions may be considered worthwhile enough to be included in the instruction list. The final choice of whether to include the skip-type instructions would depend on the cost of implementing them on a particular machine and how much time they saved.

The skip on negative storage (son x) command is the simplest to implement. The two possibilities involve making a test on bit zero of the memory buffer register or interchanging the memory buffer register and the accumulator, making the test on bit zero of the accumulator and then bringing the original contents of the accumulator, which would now be in the memory buffer register, back to the accumulator. These two methods are shown in Figure 15.

The skip on accumulator greater than storage (sag x) command is a little more complicated to implement. As shown in Figure 16 the difference between the c(x) and the accumulator must be formed and the sign of the difference must be checked. If the sign of c(x) and c(ac) are different, then overflow after addition is possible. However, the addition is not necessary since in this case an examination of the sign of either register will suffice to determine whether the condition for skipping the next instruction is met.



(a) With Circuitry to Test the Sign of the MBR



(b) With Circuitry to Interchange the AC and MBR

Fig. 15 Two Methods of Implementing the son x Command

1. $c(x) \leqslant 0$

```
son x
tra ⟶ 18 μsec.
 ↓
12 μsec.
```

2. $c(x) \leqslant$ constant

```
cla
add x
sag (constant
tra ⟶ 42 μsec.
 ↓
36 μsec.
```

3. $c(ac) \leqslant$ constant

```
sag (constant
tra ⟶ 18 μsec.
 ↓
12 μsec.
```

4. $|c(ac)| \geqslant$ constant

```
trn .+2
com
sag (constant
tra ⟶ 42 μsec.
 ↓
36 μsec.
```

5. $c(ac) \geqslant c(x)$

```
sag x
tra ⟶ 18 μsec.
 ↓
12 usec.
```

6. $c(x) = 0$

```
soz x
tra ⟶ 18 μsec.
 ↓
12 μsec.
```

7. $c(x) \geqslant c(y)$

```
cla
add x
sag y
tra ⟶ 42 μsec.
 ↓
36 μsec.
```

8. $|c(x)| \geqslant$ constant

```
cla
add x
trn .+2
com
sag (-constant
tra ⟶ 66 μsec.
 ↓
60 μsec.
```

9. $c(ac) \leqslant 0$

```
trn ⟶ 6 μsec.
 ↓
12 μsec.
```

Table XIV Coding of Decisions Using Skip-Type Instructions

	Total Worst-Case Time Saved		
	Using Skip Type Inst. (μsec.)	Using lda And sub Inst. (μsec.)	Using Both Skip Type And lda And sub Inst. (μsec.)
1. $c(x) \leqslant 0$	810	540	810
2. $c(x) \leqslant$ constant	138	276	414
3. $c(ac) \leqslant$ constant	156	0	156
4. $\lvert c(ac) \rvert \geqslant$ constant	72	0	72
5. $c(ac) \geqslant c(x)$	126	84	126
6. $c(x)=0$	90	60	90
7. $c(x) \geqslant c(y)$	27	36	63
8. $\lvert c(x) \rvert \geqslant$ constant	12	24	36
9. $c(ac) \leqslant 0$	0	0	0
Total Time Saved	1431 μsec.	1020 μsec.	1767 μsec.

Table XV Time Saved For Each Type of Decision Using Skip Type Instructions

	Worst-Case Time Saved (μsec.)
son x	810
sag x	531
soz x	90

Table XVI Total Time Saved By Each Instruction Using Skip-Type Instructions

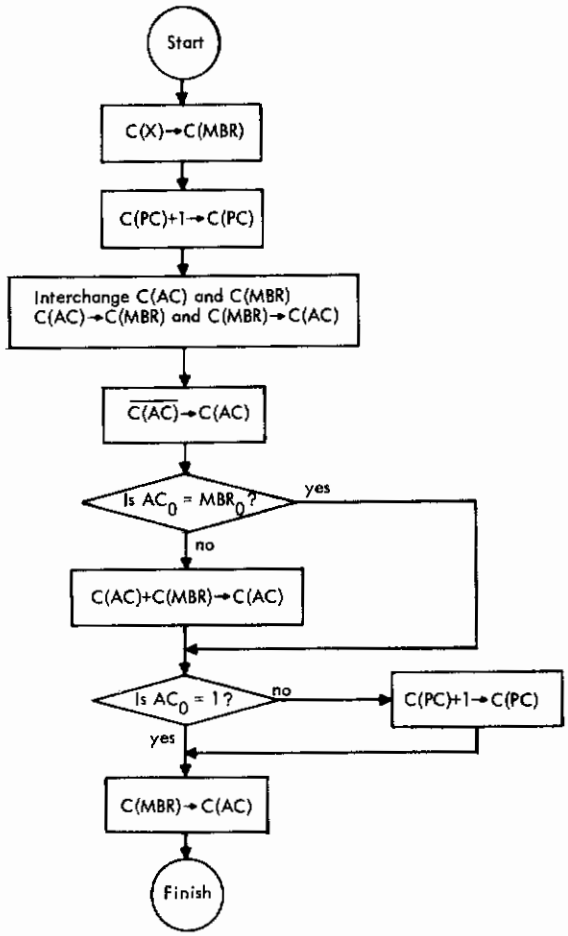


Fig. 16 Method of Implementing the sag x Command

The skip on zero storage (soz x) command is identical to the skip on negative storage command except that the test is for every bit equal to zero rather than just bit zero. The two methods shown in Figure 15 can be used providing that the test is changed accordingly.

Since the skip on accumulator greater than storage command requires the circuitry to interchange the accumulator and memory buffer register, it would seem that the method shown in Figure 15b for the skip on negative storage and the skip on zero storage commands would be preferable due to the additional circuitry that would be required to do the tests on the memory buffer register. The circuitry to test the accumulator is already there due to the transfer negative and transfer zero commands.

8.3 Analog and Digital Inputs and Outputs

Table XVII shows the number of analog and digital inputs and outputs for the F-100A simulation program.

Analog Inputs	8
Discrete Inputs	43
Analog Outputs	37
Discrete Outputs	15

Table XVII Number of Discrete and Analog Inputs and Outputs Used for the F-100A Simulation

As in the case of decisions, individual inputs and outputs do not take very much time and, therefore, it is not expected that very much time can be saved by special-purpose instructions. However, in other real-time simulation problems or in a general-purpose simulator, they may become much more important. The method used for the TX-O program and other possible methods will be discussed in the next four sections.

8.4 Analog Inputs

For the TX-O program, all analog inputs are assumed to be connected to the input of an encoder through sample gates. A sample gate is an analog device whose output is either equal to the input or at ground potential, depending on the control input. When it is desired to bring an analog input into the computer, it is necessary to activate the proper sample gate so that the selected analog voltage appears on the input of the encoder. Then the encoder converts the analog voltage into a parallel digital number, which is, in the case of the TX-O, strobed into the live register. The method assumed here of controlling the encoder and the sample gates is that used by Binsack. A full description of the control system appears in Reference 22. To convert an analog variable into a digital number in the live register is a six-instruction sequence requiring 72 μsec.

8.5 Discrete Inputs

Discrete inputs are a type of decision since they are only interrogated with the corresponding program branch is encountered. The method used

for the TX-O program combines strobing-in the input with the branching operation. The 43 discrete inputs are assigned to three eighteen-bit registers. Any one of these three registers can be strobed into the live register by giving the in-out command associated with that register. The circuitry used is shown in Figure 17. Actually the final contents of the live register is the result of an or operation between the eighteen discrete inputs in the particular bank addressed and the corresponding bits of the previous contents of the live register. Therefore, if the live register initially contains all ones except for a zero in the position corresponding to the desired discrete input, then after the or operation between the live register and the bank of discrete inputs, the live register will contain a binary minus zero if and only if the discrete input of interest was a one. The contents of the live register is then placed in the accumulator in the same instruction and a transfer zero instruction accomplishes the branching. An explanatory note on the one's complement zero is given in Appendix 1.

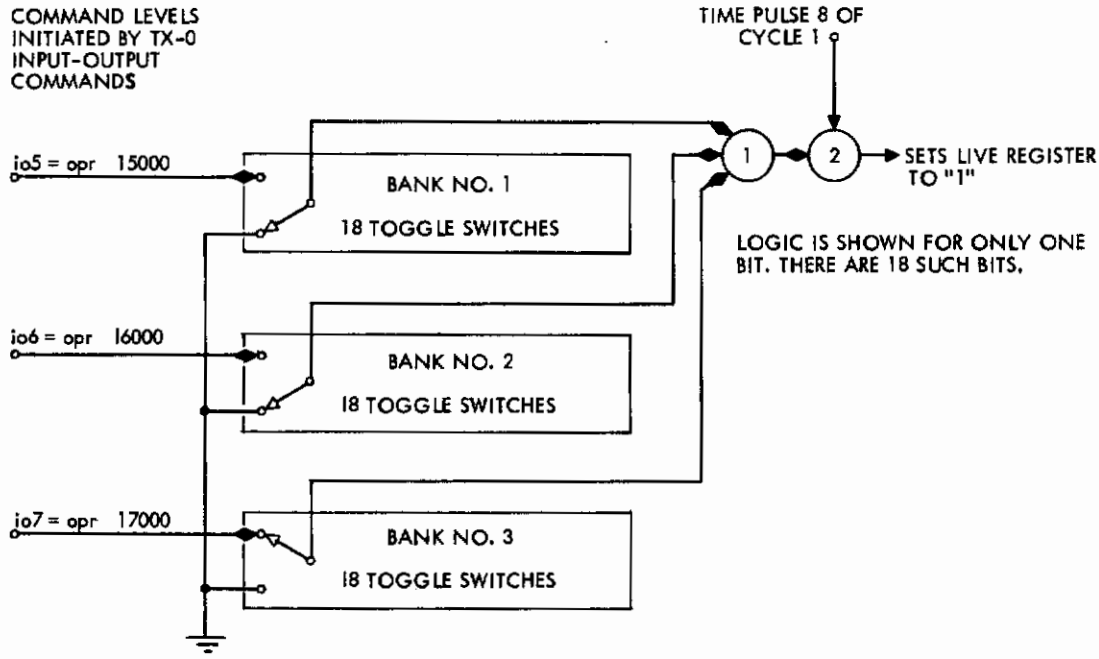


Fig. 17 Method of Handling Discrete Inputs

As an example, the following sequence will branch on bit five of bank one assuming that the command io5 will "or" bank one with the live register:

```
        llr (767777
        io5Ulac
        trz  ──►   transfers if bit 5 is a one
          │
          ▼
does not transfer if bit 5 is a zero
```

This sequence requires 30 μsec. if bit 5 is a one, and 36 μsec. if bit 5 is a zero.

UDOFT uses a memory register for each discrete input. Each discrete input is wired directly into a corresponding memory location; when the discrete input is off, the reading of the associated word from memory is inhibited. If this method were used for the TX-O, then the skip on negative storage (sonx) command could be used to save about 800 μsec. of worst-case time for discrete inputs. As the number of discrete inputs increases, this method of using one memory register for each discrete input becomes progressively more inefficient in memory utilization.

8.6 Analog Outputs

As with analog inputs, the TX-O program for analog outputs assumes that the TX-O is connected to the system designed by Binsack and described in Reference 22. For analog outputs, the live register is connected directly to a decoder. The output of the decoder is connected to a number of storage gates. A storage gate is an analog device whose output is set to the input voltage level during a charge cycle. During the hold cycle, the output of the storage gate remains fixed at this voltage level independent of the input voltage. The charge and hold cycles of the storage gate are controlled by a control input. The system designed by Binsack controls the decoder and the storage gates so that a four-instruction sequence requiring 48 μsec. decodes a digital variable and stores the result on the output of a particular storage gate.

8.7 Discrete Outputs

The fifteen discrete outputs used in the TX-O program are stored in one memory register. This register is set to zero at the beginning of every 50 msec. solution cycle. When it is desired to set a particular discrete output to one, the contents of the discrete output memory register are brought

into the accumulator, a one is placed in the proper bit, and the revised discrete output word is stored back in memory. This four-instruction sequence requires 48 μsec. At the end of each 50 msec. cycle, the register containing the discrete outputs is placed in the live register and, with the use of an in-out command, the discrete output bits are used to set an external flip-flop register. The indicator lights on this external register are the discrete output indicators that appear on the pilot's and instructor's consoles. This external circuitry is shown in Figure 18.

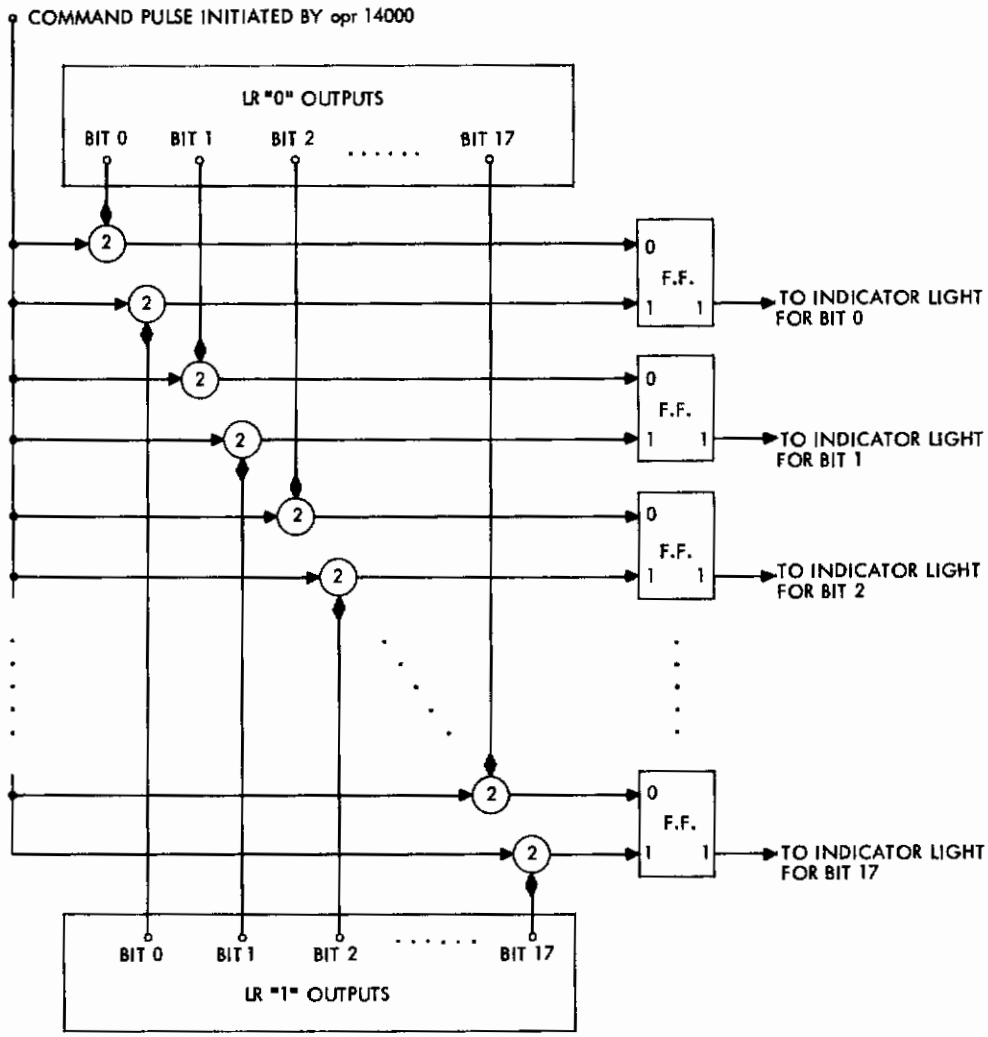


Fig. 18 Method of Handling Discrete Outputs

This method is fairly efficient in running time for this particular problem because of certain characteristics of the method and the problem. Whether a discrete output is represented by a zero or a one in the core memory register

is completely arbitrary, since the indicator light can be driven off either the one output or the zero output of the external flip-flops. When the program requires that a discrete output be set in a worst-case routine, then the convention used is such that the corresponding location in the discrete output register should be set to a zero. Because every position of the core memory register is initially set to zero every cycle, no action is required by the program in this case. This procedure will minimize the worst-case running time with respect to discrete outputs. It is required here that once the decision is made to set a discrete output to a one, then there must be no other subsequent section in the program that could require the discrete output to be zero. This requirement is met in the F-100A problem; but in a more complicated problem, it might be necessary to actually set discrete output bits to zero as well as to one.

UDOFT uses a more general method that can be extended easily to larger and more complicated problems. An addressable command is defined that examines the sign bit of the accumulator and sets a discrete output to zero if the accumulator is positive or to one if the accumulator is negative. The address section of the instruction addresses the particular discrete output to be set.

The method used for the TX-O is economical in worst-case running time and does not require any special-purpose instructions. For a problem where discrete outputs are more complicated, then the special instruction used by UDOFT would be recommended.

8.8 Summary of Conclusions for Decisions and Inputs and Outputs

For the F-100A problem, no individual additional commands save enough time for decisions, inputs, and outputs to be recommended. The load accumulator and subtract instructions are considered worthwhile and have already been discussed in Chapter V. It should be pointed out that the 1.020 msec. saved in Table XV using these instructions has already been included in the 11.190 msec. saving cited in Table VII.

For a general-purpose real-time simulation facility or for larger and more complicated problems, individual special-purpose commands might be justified.

CHAPTER IX

HIGH-SPEED MULTIPLICATION

9.1 Introduction

From the previous chapters, it has been seen that a good order code for a complex real-time simulation problem is general-purpose with just a few special-purpose instructions. There is one important aspect of the problem that has not yet been discussed, the multiply instruction. For the original TX-O program, a 25 μsec., non-addressable multiply was assumed. This multiply instruction, with the associated load live register instruction that accompanied every multiply, accounts for 14.541 msec. of the running time in the worst case or 27% of the worst-case running time. In Chapter V the savings in time and memory using a 25 μsec. addressable multiply command was found to be 4.716 msec. and 453 registers, respectively. This multiply was modeled after a 25 μsec. addressable multiply available as an option on the Digital Equipment Corporation's PDP-1. Neither of these multiply orders corresponds very closely to the one scheduled for eventual installation on the TX-O. Present plans for the TX-O call for a non-addressable multiply command requiring approximately 60 μsec. maximum.

In this chapter, a survey of current methods of high-speed multiplication will be presented. To provide a quantitative basis on which to select the multiply logic for a simulation computer, a method of comparing the relative speed and complexity of various multiply techniques is outlined.

9.2 Methods of High-Speed Multiplication

Methods for achieving high-speed multiplication fall into two distinct categories. The first category is concerned with the method of multiplication itself. This includes such factors as the multiplication algorithm used, coding of the multiplier bits, number of multiplier bits examined at each cycle, etc. The second category is concerned with the methods of speeding up the process of addition. Since every multiply method involves some additions, the time required for multiplication depends quite strongly on the addition time.

In 1959, Lyon,[16] working at the M.I.T. Electronic Systems Laboratory, completed a study of high-speed addition and multiplication methods. The different methods investigated by Lyon were:

1) Full add and shift
2) Full add and shift using carry completion detection
3) Half add and shift using carry storage
4) Series-parallel multiplication
5) Ripple multiplication

Ripple multiplication is a form of series-parallel multiplication using pulse logic. Although ripple multiplication was the fastest method investigated by Lyon, the disadvantages of pulse logic are felt to outweigh the speed advantage.

There is another method of multiplication, not included by Lyon, that has achieved some popularity in the current literature.[17, 18, 19] This method, called "Recoding of the Multiplier" by the University of Illinois will be described in detail in the following section.

9.3 Recoding of the Multiplier

If A is the multiplier and B the multiplicand, a common procedure for obtaining the product is by successive additions of the multiplicand to a running sum, which is shifted after each step to correspond to the binary bit of the multiplier controlling the next addition. We have:

$$B\,A = B(a_k 2^k + a_{k-1} 2^{k-1} + \ldots + a_1 2 + a_0) \tag{9.1}$$

where the coefficients a_j are either <u>one</u> or <u>zero</u> depending on the bit of the multiplier A.

The method to be described in this section is based on the fact that in the multiplier A, a string of n adjacent binary ones with numerical value:

$$2^{n+i-1} + a^{n+i-2} + \ldots + 2^{i+1} + 2^i \tag{9.2}$$

can also be written

$$2^{n+i} - 2^{i} . \tag{9.3}$$

In other words, instead of n successive computer additions of the multiplicand B to the running sum, the same result can be achieved by one addition and one subtraction. The process can be further extended to take into account single included zeros in a string of adjacent binary ones:

$$2^{n+i-1} + 2^{n+i-2} + \ldots + 2^{j+i+1} + 2^{j+i-1} + \ldots + 2^{i} . \tag{9.4}$$

Here the 2^{j+i} term is missing and this sum can be written as:

$$2^{n+i} - 2^{j+i} - 2^{i} . \tag{9.5}$$

This approach can obviously be extended for any number of zeros within a string of adjacent binary ones, however, the method to be described here will recognize only single included zeros.

As an example, consider the following multiplier:

0 0 1 0 1 1 1 1 0 1 0 0 1 0 1 1 0

Two bits of the multiplier (starting with the least significant pair) will be examined on each cycle, and the decision will be made as to whether the multiplicand should be added to the partial product, subtracted from the partial product, or neither. In any case, the multiplier and the partial product are both shifted to the right one position and the process repeated until the entire multiplier has been examined. The notation used will be that a symbol, +, -, or •, written below a multiplier bit will specify the action to be taken when that particular bit of the multiplier has been shifted into the least significant position. A plus sign will indicate that the multiplicand will be added to the partial product; a minus sign will indicate that the multiplicand will be subtracted from the partial product, and a dot will indicate that no action will be taken on that cycle. Using the identities in equations 9.1 through 9.5, the action taken upon each bit of the above multiplier starting with the least significant bit would be:

0 0 1 0 1 1 1 1 0 1 0 0 1 0 1 1 0
• + • - • • • - • + • + • - • - •

In its most basic form, the method requires that the two least significant bits of the multiplier be examined at each cycle. The method also requires both add and subtract logic. Figure 19 shows in schematic form the logical connections between the various arithmetic registers. The names of the arithmetic registers used are those on the TX-O. The logic is set up to multiply the contents of the memory buffer register by the contents of the live register leaving the result in the accumulator, which is cleared at the start of multiplication.

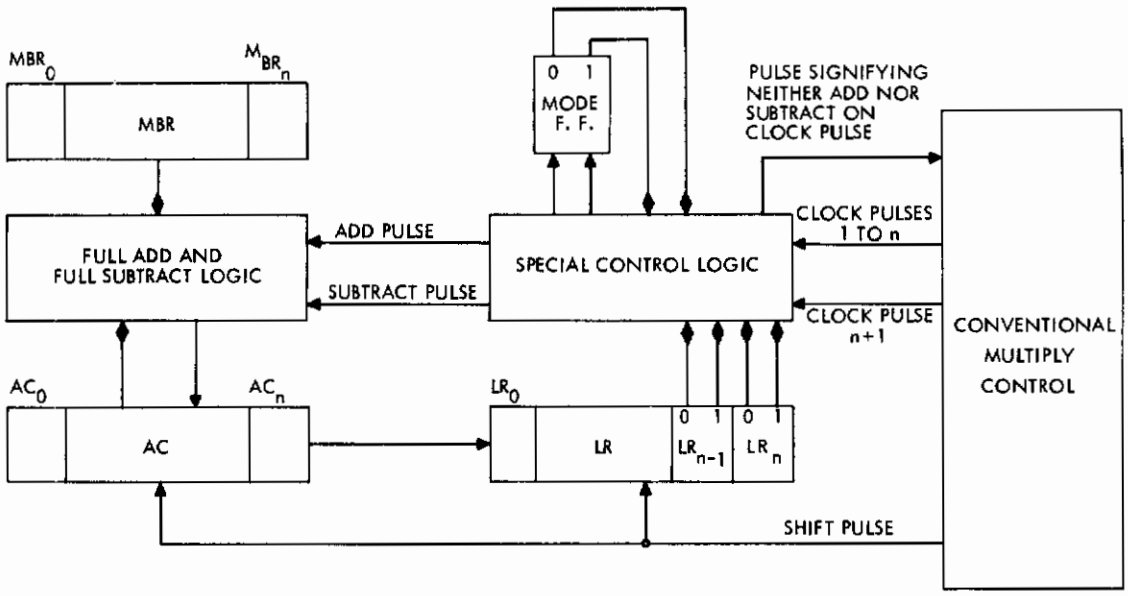


Fig. 19 Recoding of Multiplier Method of Multiplication

In Figure 19 the box called Conventional Multiply Control contains the logic which is common to practically all multiplication schemes. It includes a counter to terminate the multiplication after all bits have been multiplied and it includes the circuitry to distribute the clock and shift pulses. The box called Special Control Logic contains the circuitry necessary to examine the last two bits of the live register and the present status of the mode flip-flop in order to generate the add, subtract, or neither-add-nor-subtract pulse. The mode flip-flop is really part of the Special Control Logic and is shown separately only to emphasize the fact that it is the only extra flip-flop needed in addition to those in the three arithmetic registers and the Conventional Multiply Control.

In the following discussion, a string of ones will mean a group of binary ones that are either adjacent or separated at any point by at most one binary

zero. A string must have at least two adjacent binary ones at the right-hand end. With this definition, the sequence, 1 0 1 0 1 1 1, is a string, but the sequence, 1 0 1 0 1, is not, since there is only one binary one at the right-hand end.

The two states of the mode flip-flop are defined as:

M = 0 Not within a string of ones.

M = 1 Within a string of ones.

The mode flip-flop is initially set to zero at the start of the multiplication and this setting prevails during the first step. The truth table relating the succeeding states of the mode flip-flop and the last two bits of the live register with the resultant action is shown in Table XVIII.

M \ $LR_{n-1}LR_n$	00	01	11	10
0	no action	add leave M = 0	subtract set M = 1	no action
1	add set M = 0	no action	no action	subtract leave M = 1

Table XVIII Truth Table for Recoded Multiplier Method

After each examination, and the subsequent add or subtract, if any, both the live register and the accumulator are shifted to the right one position and the process is repeated.

In order for the multiplication to proceed more rapidly than the conventional full add and shift method, it is necessary to perform the shift immediately if neither an add nor subtract are to be performed on a particular cycle.

If a pulse on the add line is called A, a pulse on the subtract line is called S, and a pulse on the neither-add-nor-subtract line is called N, then from Table XVIII it can be seen that the Boolean expressions required are:

$$A = (\overline{M} \cdot \overline{LR}_{n-1} \cdot LR_n + M \cdot \overline{LR}_{n-1} \cdot \overline{LR}_n) \cdot (\text{clock pulse}) \qquad (9.6)$$

$$S = (\bar{M} \cdot LR_{n-1} \cdot LR_n + M \cdot LR_{n-1} \cdot \overline{LR}_n) \cdot (\text{clock pulse}) \quad (9.7)$$

$$N = (\bar{M} \cdot \overline{LR}_{n-1} + M \cdot LR_{n-1}) \cdot (\text{clock pulse}) = \overline{A + S} \quad (9.8)$$

$$M = S \quad (9.9)$$

$$\bar{M} = A \quad (9.10)$$

Figure 20 shows the circuitry required to realize these signals.

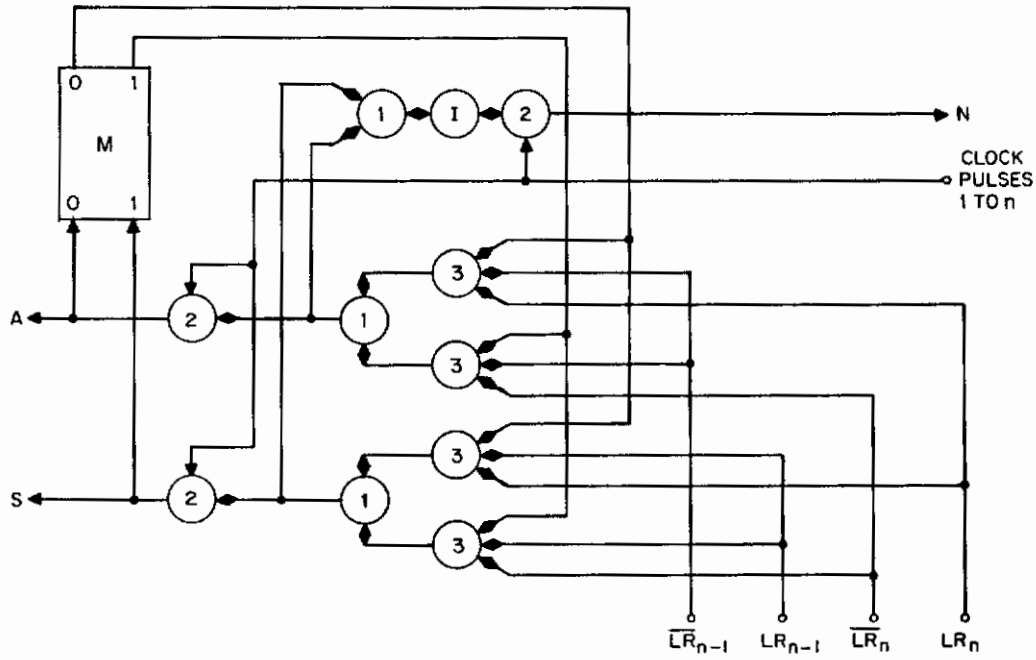


Fig. 20 Special Control Logic for Recoding of Multiplier Method

There are two other special-purpose circuits necessary. One performs the special handling necessary for the last clock pulse and the other is a special shift circuit for bit zero of the accumulator.

If the most significant bits of the multiplier are part of a string of ones, then an add should be performed after the last shift as though the multiplier had one more bit equal to a zero. This add does not come about due to the state of bits LR_{n-1} and LR_n, but rather due to the recognition of the last clock pulse and the state of the mode flip-flop, M. From the assignment of the states of flip-flop M, it can be seen that when the last pulse appears

on the separate line shown in Figure 19 and M is in state one, then the memory buffer register should be added to the accumulator after the last shift. The Boolean function required here is simply:

$$A = M \cdot (\text{last clock pulse}) \tag{9.11}$$

The other special circuit is necessary because bit zero of the accumulator could be in the one state either due to overflow after adding or due to the accumulator being negative after subtraction. The action required on the shift pulse is different in these two cases. If bit zero of the accumulator is a one due to overflow after adding, then we wish to replace bit zero with a zero when shifting. If bit zero of the accumulator is a one due to subtraction, then we wish to keep the accumulator negative by leaving bit zero in the one state after shifting. The mode flip-flop again contains the information as to whether the accumulator should be positive or negative after the shift. If M = 0, then the last arithmetic operation was an addition and the accumulator should be positive. If M = 1, then the last arithmetic operation was a subtraction and the accumulator should be negative. In this last case, the accumulator already is negative so that bit zero need not be changed. The only input to bit zero that is necessary is the set to zero input:

$$\overline{AC}_0 = \overline{M} \cdot (\text{shift pulse}). \tag{9.12}$$

A closer investigation into the implications of the partial product in the accumulator being either positive or negative at various stages of the multiplication leads to a much more complicated problem. Up to now it has been tacitly assumed that the number system being used is the 1's complement representation of negative numbers used on the TX-O. Using this number system, almost all multiplication methods require that the multiplier and the multiplicand be positive. The sign of the product is stored in the Conventional Multiply Control and is affixed to the product after the multiplication is completed. If the partial product can be either positive or negative, then the least significant bit of the accumulator that is shifted into the most significant bit of the live register is very difficult to interpret. In general, when using the 1's complement number system, the less significant part of the product appearing in the live register after the multiplication is completed is meaningless. In other words, this method of multiplication, when used with the 1's complement number system, yields only a single register product.

For the TX-O simulation program, a single-length product is all that is necessary. This is true because of the way variables are scaled. When a variable takes on its maximum value, the machine representation of the variable contains a one in the most significant bit. After two variables are multiplied, it is possible for the most significant bit of the accumulator to be a one if both of the variables were close to their maximum value. In this case, it is not possible to use any of the bits of the less significant half of the product, since shifting them into the accumulator would cause the loss of significant bits at the left-hand end of the accumulator. For this reason, the least significant part of the double-length product after multiplication is never used in the TX-O program and a special-purpose multiply that resulted in only a single-length product would be acceptable.

This does not mean that the contents of the live register before multiplication would not be destroyed by the multiplication process. The multiplier is placed in the live register at the start of the multiplication process thereby destroying the previous contents of the live register, and the best that can be done, without providing an additional 18 bit arithmetic register, is to have the multiplier available in the live register after the multiplication.

The University of Illinois Graduate College Digital Computer Laboratory[17] used this multiplication method in the design of a very high-speed computer. One of the results of this study was that if the 2's complement representation of negative numbers is used, then this multiplication method results in a correct double-length product. It was mainly for this reason that they adopted the 2's complement system for their computer.

Lyon has shown that when the 2's complement system is used, both the multiplier and the multiplicand can have arbitrary signs and the multiplication will still be correct.

This property of the 2's complement number system has a very interesting effect on the total operation time for an addressable multiply instruction. In a computer using magnetic core memory with a memory cycle time of t_{mc}, the first two parts of the operation of an addressable multiply command would be getting the instruction from memory and getting the multiplicand from memory. Currently-used types of magnetic core memory use destructive read-out and, therefore, as soon as the memory word is in the memory buffer register, it must be read back into memory to achieve the non-destructive read characteristic of most computers. The word from memory is actually in the memory buffer register after only half of the memory cycle time has

elapsed. While the word is being read back into memory, it can be used provided that the memory buffer register is not changed. If the 2's complement number system is used, the multiplication could start after only $(3/2)t_{mc}$ has elapsed, since there is no need to change the memory buffer register before the multiplication begins. If the 1's complement number system or an absolute magnitude number system is used, however, the actual multiplication cannot begin until $2t_{mc}$ has elapsed due to the fact that the sign of the memory buffer register may have to be changed and this cannot be done before the word is read back into memory. This advantage of the 2's complement number system will be apparent later in the chapter when a comparison of the operating time for a number of different multiplication methods is presented.

The longest multiply time using the recoded multiplier method will result from a multiplier of the form:

1 0 1 0 1 0 . . . 1 0 1 0 1 .

The notation used for multiply time in the remainder of the chapter will be:

t_{max}	maximum multiply time including memory accesses for an addressable multiply command.
t_{avg}	average multiply time
t_{ha}	half-add time
t_{cp}	carry propagate and carry addition time for an n bit register.
t_s	shift time
t_c	time required to complement a flip-flop
t_{mc}	memory cycle time
n	number of bits excluding sign bit

The maximum and average multiply times for the recoded multiplier method on a machine using 1's complement or absolute magnitude negative number representation are given by the University of Illinois as:[17]

$$t_{max} = \frac{n}{2}(t_{ha} + t_{cp}) + nt_s + 2t_{mc} + 2t_c \qquad (9.13)$$

$$t_{avg} = \frac{n}{3}(t_{ha} + t_{cp}) + nt_s + 2t_{mc} + 2t_c \qquad (9.14)$$

The $2t_c$ factor on the end is the time required to set the sign of both factors positive at the beginning and to correct the sign of the product at the end.

Using the 2's complement system the corresponding times would be:

$$t_{max} = \frac{n}{2}(t_{ha} + t_{cp}) + nt_s + \frac{3}{2}t_{mc} \qquad (9.15)$$

$$t_{avg} = \frac{n}{3}(t_{ha} + t_{cp}) + nt_s + \frac{3}{2}t_{mc} \qquad (9.16)$$

Actual values for the various circuit times will be used later in the chapter in order to compare these methods to those already discussed by Lyon.

9.4 Recoding of the Multiplier with Multiple Shift

As pointed out by Smith and Weinberger,[18] the average multiply time can be decreased by being able to shift the accumulator and live register by more than one binary position on each cycle. They suggest incorporating circuitry to shift up to four places, in which case, the four least significant bits of the multiplier must be examined. In this section, the problem of performing either one or two shifts on each cycle will be examined. This additional complexity decreases both the average and the maximum multiply time. Providing the circuitry to shift more than two places on each cycle decreases the average multiply time, but does not decrease the maximum multiply time any further.

Figure 21 shows a few bits of a register that can be shifted either one or two binary positions on each cycle depending on which control line is pulsed.

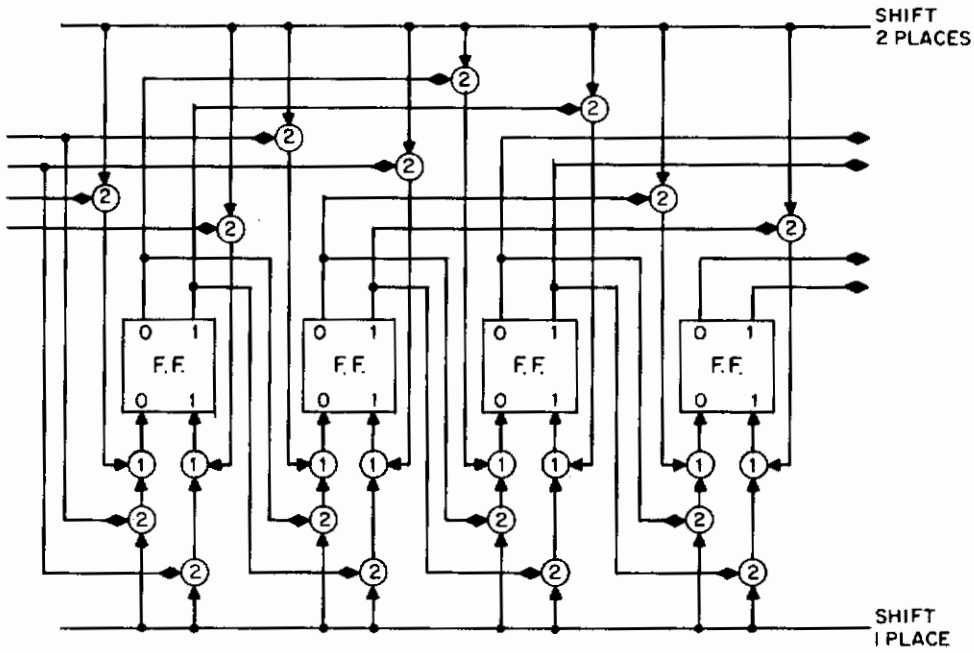


Fig. 21 One or Two Position Shift Register

In order to determine whether to shift one position or two on a particular cycle, a new truth table is needed.

M \ LR_{n-1} LR_n	00	01	11	10
0	shift 2	shift 2	shift 2	shift 1
1	shift 2	shift 1	shift 2	shift 2

Table XIX Truth Table For Shifting Multiplier

From Table XIX the required signals for the shift lines are:

$$\text{Shift 1 Pulse} = (\overline{M} \cdot LR_{n-1} \cdot \overline{LR}_n + M \cdot \overline{LR}_{n-1} \cdot LR_n) \cdot (\text{shift pulse}) \qquad (9.17)$$

$$\text{Shift 2 Pulse} = \overline{(\overline{M} \cdot LR_{n-1} \cdot \overline{LR}_n + M \cdot \overline{LR}_{n-1} \cdot LR_n) \cdot (\text{shift pulse})} \qquad (9.18)$$

These signals can be formed in the same way as the add and subtract pulses in Figure 20.

Using this circuitry, the longest multiply time again is caused by alternate ones and zeros in the multiplier:

1 0 1 0 1 0 . . . 1 0 1 0 1 .

Now, however, the zeros are skipped and therefore the maximum multiply time using an addressable multiply with the 1's complement number system is:

$$t_{max} = \frac{n}{2}(t_{ha} + t_{cp} + t_s) + 2t_{mc} + 2t_c \qquad (9.19)$$

Using 2's complement number system the maximum multiply time is:

$$t_{max} = \frac{n}{2}(t_{ha} + t_{cp} + t_s) + \frac{3}{2}t_{mc} \qquad (9.20)$$

It should be pointed out that this is also the maximum multiply time no matter how many shifts can be performed on any cycle, since alternate zeros and ones are always the worst case.

9.5 High-Speed Carry Propagation

Lyon[16] describes a number of conventional methods of carry propagation. For this study, two methods will be used. The first, which will be called the conventional carry system, is characterized by the fact that the carry pulse, in propagating from bit to bit, goes through a flip-flop at each stage. Because of the delay through the flip-flop, this method is rather slow For this study t_{cp} will be used to denote the conventional carry propagate and process time.

The second method of carry propagation, which will be called the high-speed carry system, is characterized by the fact that in propagating through the register, the carry goes through only one transistor per stage, and is therefore, delayed much less than the conventional carry propagate, which requires additional flip-flop action. Using a form of high-speed carry, Salter[20] at the Argonne National Laboratory reports a maximum add time of 0.23 μsec. for a 67 bit addition. For this study, t'_{cp} will be used to denote the high speed carry propagate and process time.

9.6 Comparison of Multiply Times

In making a comparison between different methods of multiplication, the two most important factors are speed and cost. In this section the relative speeds of the methods discussed by Lyon and the methods introduced in this chapter will be discussed. In the following section, an attempt will be made to compare the logical complexity of the different methods.

In order to compare multiply times, representative values of the response times of the various logical elements have been chosen. The circuitry is all assumed to be 5 mc. circuitry of the TX-O, TX-2, PDP-1 type. The memory cycle time is that found in the Digital Equipment Corporation's PDP-1.

t_{ha} = 0.2 μsec. half-add time

t_{cp} = 0.55 μsec. for n = 17 carry propagate and carry addition time

t'_{cp} = 0.06 μsec. for n = 17 carry propagate and carry addition time for high-speed carry

t_s = 0.2 μsec. shift time

t_c = 0.2 μsec. time required to complement a flip-flop

t_{mc} = 5.0 μsec. memory cycle time

n = 17 number of bits excluding sign bit

kn = 4.4 for n = 17* average maximum carry length for an n bit word

Table XX presents the maximum multiplication times for the methods described by Lyon and those described in this chapter. The methods using carry completion and recoding of the multiplier with shift of one or two position require an asynchronous computer in the sense that when an add is completed or if no add is performed on a particular cycle, then the pulse on the

---

* Hendrickson[21] shows that for n > 10 the average maximum carry can be approximated by the formula, $kn = \log_2(\frac{5n}{4})$

	Multiplication Method	Number System *	Maximum Multiply Time	(μsec)
1	full add and shift	1	$n(t_{ha} + t_{cp} + t_s) + 2t_{mc} + 2t_c$	26.55
2	full add and shift	2	$n(t_{ha} + t_{cp} + t_s) + \frac{3}{2} t_{mc}$	23.65
3	full add and shift with carry completion	1	$n(t_{ha} + k\ t_{cp} + t_s) + 2t_{mc} + 2t_c$	20.60
4	full add and shift with high-speed carry	1	$n(t_{ha} + t'_{cp} + t_s) + 2t_{mc} + 2t_c$	20.60
5	recoding of the multiplier **	1	$\frac{n}{2}(t_{ha} + t_{cp}) + nt_s + 2t_{mc} + 2t_c$	20.55
6	recoding of the multiplier ** shift 1 or 2 positions per cycle	1	$\frac{n}{2}(t_{ha} + t_{cp} + t_s) + 2t_{mc} + 2t_c$	18.75
7	carry storage	1	$n(t_{ha} + t_s) + t_{cp} + 2t_{mc} + 2t_c$	17.75
8	series-parallel	1	$2\,n t_{ha} + t_{cp} + 2\,t_{mc} + 2t_c$	17.75
9	recoding of the multiplier	2	$\frac{n}{2}(t_{ha} + t_{cp}) + nt_s + \frac{3}{2} t_{mc}$	17.65
10	recoding of the multiplier ** with carry completion	1	$\frac{n}{2}(t_{ha} + k\,t_{cp}) + nt_s + 2t_{mc} + 2t_c$	17.40
11	recoding of the multiplier shift 1 or 2 positions per cycle	2	$\frac{n}{2}(t_{ha} + t_{cp} + t_s) + \frac{3}{2} t_{mc}$	15.85
12	recoding of the multiplier ** shift 1 or 2 positions per cycle with carry completion	1	$\frac{n}{2}(t_{ha} + k\,t_{cp} + t_s) + 2t_{mc} + 2t_c$	15.60
	Ripple multiplication (pulse logic)	1	$(n-1)\,t_{ha} + \frac{3(2n-1)}{2n} t_{cp} + 2t_{mc} + 2t_c$	14.65
13	recoding of the multiplier shift 1 or 2 positions per cycle with carry completion	2	$\frac{n}{2}(t_{ha} + k\,t_{cp} + t_s) + \frac{3}{2} t_{mc}$	12.70
14	recoding of the multiplier shift 1 or 2 positions per cycle with carry storage	2	$\frac{n}{2}(t_{ha} + t_s) + t_{cp} + \frac{3}{2} t_{mc}$	11.75

* 1 - 1's complement or absolute magnitude number system
2 - 2's complement number system
** results in a single length product

Table XX Maximum Multiply Time

shift line must appear immediately. All of the values in Table XX are not arrived at directly by using the response times for the various logical elements. The fact that the maximum repetition frequency for the flip-flop is 5 mc. limits the time between successive operations in some cases. For example, using full add and shift with carry completion, the maximum multiply time is:

$$t_{max} = n(t_{ha} + k\,t_{cp} + t_s) + 2t_{mc} + 2t_c \qquad (9.21)$$

On the average, the carry after each add will be completed after an elapsed time of:

$$k\,t_{cp} = \frac{4.4}{17}\,(0.55) = 0.14\ \mu\text{sec.} \qquad (9.22)$$

Theoretically it would then be possible to shift the accumulator and live register after this time has elapsed. This is not true for the 5 mc. circuitry used, however, since this could result in a flip-flop being triggered by two signals less than 0.2 μsec. apart. Therefore, it is necessary to delay any operation that could effect the same flip-flop within less than 0.2 μsec. In the example of equation 9.21, this would mean that 0.2 μsec. was used rather than 0.14 μsec. for $k\,t_{cp}$.

The information in Table XX will be used with the complexity factor given in the next section to arrive at a quantitative measure for selecting a multiplication method.

9.7 Comparison of Complexity

The relative complexity of multiplication methods is more difficult to define than relative operating time. When a computer manufacturer designs a multiplier for his computer, the additional complexity is reflected only in the additional total cost. Comparing the total cost of various machines is not helpful, since the cost takes into account many extraneous factors such as profit structure and the number of machines that the manufacturer expects to build.

The only completely valid means of comparison is to design a multiplier for a particular computer using each type of multiplication method in question. The measure of logical complexity could be, perhaps, the number of transistors and diodes employed.

Such a detailed design survey was outside the scope of the present study, but in order to demonstrate the methodology suggested, an attempt was made to associate a complexity index with all of the multiplication methods shown in Table XX except for ripple multiplication. Since ripple multiplication uses pulse logic, it was felt that the basic arithmetic unit would be so different from the arithmetic unit used by the other methods that a meaningful comparison would be impossible.

The complexity index, in each case, is proportional to the number of transistors and diodes used for the multiply circuitry. The exact proportionality factor is not known, however. The first assumption made was that the control circuitry used for the full add and shift method (shown on the top line of Table XX) is common to all multiplication methods investigated. Under this assumption, the complexity index of any multiplication method would be the complexity index of the full add and shift method plus the complexity index for any additional circuitry needed to implement the particular multiply scheme.

This additional circuitry is shown in some detail, either in this chapter or in Lyon's study, for all the methods of multiplication shown in Table XX. This circuitry is not designed in such detail as to show individual transistors and diodes, but rather in terms of flip-flops, "and" gates, and "or" gates. For simplicity, the measure of complexity is taken equal to six times the number of additional flip-flops plus the number of inputs to "and" and "or" gates. Since the full-add-and-shift multiplier corresponds quite closely to the high-speed multiplier available as an option on Digital Equipment Corporation's PDP-1, and since this computer is representative of the class of machines for which these multiply methods were designed, the multiply control logic used by PDP-1 was chosen as the model for the basic control logic. On the basis of discussions with Digital Equipment Corporations's engineers concerning the design of their multiply option, it was decided to assign a complexity index of 240 to the basic full add and shift circuitry. Therefore, with this rather burdensome number of assumptions, the complexity index, I, used to compare the different multiplication methods is:

$$I = 240 + N + 6F \tag{9.23}$$

where:

I = Complexity Index

N = Number of imputs to gates in the additional circuitry

F = Number of flip-flops in addition to the 3n in the arithmetic section and those already included in the 240 for basic multiply control

As an indication of the relative magnitude of the complexity index, it has been estimated that an 18 bit accumulator with the circuitry for full add and subtract, the circuitry to shift right or left one position, and the circuitry to perform logical addition and logical multiplication, would have a complexity index of approximately 360.

Table XXI shows N, F, and I for all the methods of multiplication shown in Table XX with the exception of ripple multiply. The maximum multiply time is also repeated for convenience.

This same information is also plotted in Figure 22. Notice that a straight line very nearly goes through all of the points representing the 2's complement method, and that a straight line displaced by 2.9 μsec. to the right of the 2's complement line fits the 1's complement points fairly well. This 2.9 μsec. is the difference between two identical methods of multiplication using the two different number systems.

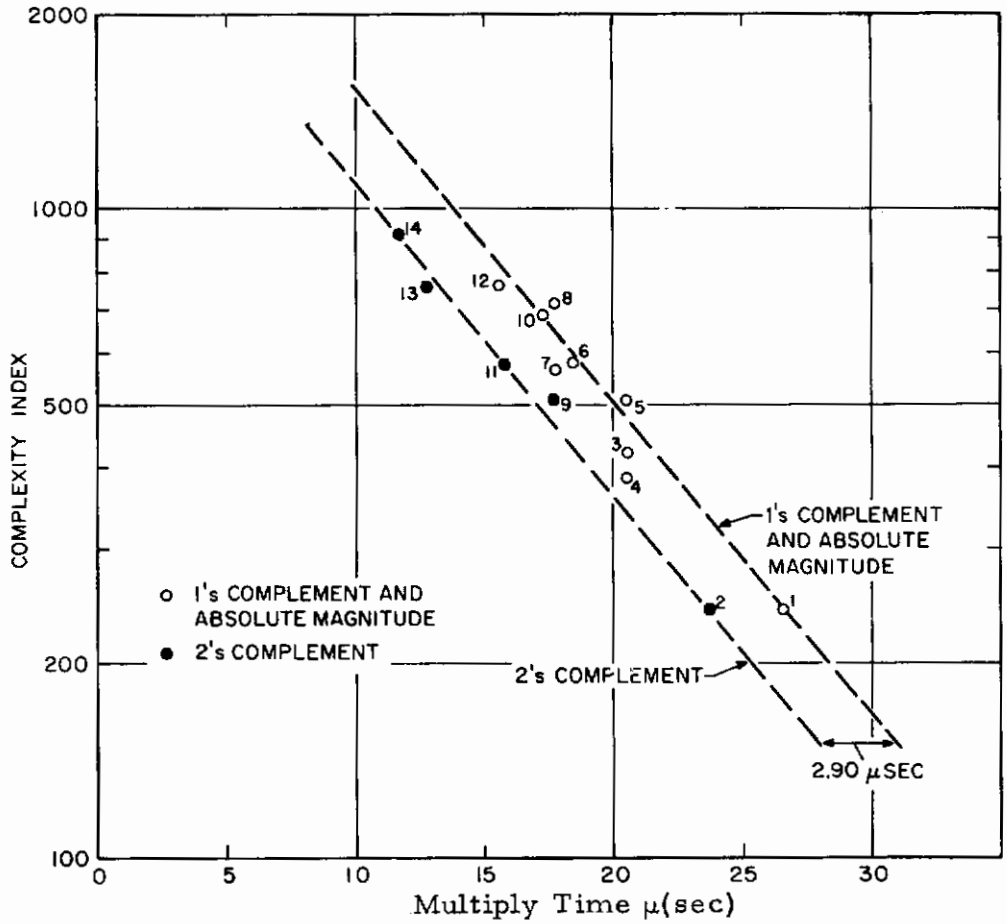


Fig. 22 Complexity Index and Multiply Time for Various Multiply Methods

	Multiplication Method	Number System *	N	F	I	t_{max} (μsec.)
1	full add and shift	1	0	0	240	26.55
2	full add and shift	2	0	0	240	23.65
3	full add and shift with carry completion	1	180	0	420	20.60
4	full add and shift with high-speed carry	1	144	0	384	20.60
5	recoding of the multiplier	1	261	1	507	20.55
6	recoding of the multiplier shift 1 or 2 positions per cycle	1	333	1	579	18.75
7	carry storage	1	216	18	564	17.75
8	series-parallel	1	144	54	708	17.75
9	recoding of the multiplier	2	261	1	507	17.65
10	recoding of the multiplier with carry completion	1	441	1	687	17.40
11	recoding of the multiplier shift 1 or 2 positions per cycle	2	333	1	579	15.85
12	recoding of the multiplier shift 1 or 2 positions per cycle with carry completion	1	513	1	759	15.60
13	recoding of the multiplier shift 1 or 2 positions per cycle with carry completion	2	513	1	759	12.70
14	recoding of the multiplier shift 1 or 2 positions per cycle with carry storage	2	549	19	903	11.65

* 1 - 1's complement and absolute magnitude number system
  2 - 2's complement number system

Table XXI Complexity Index

It should be emphasized that these results are only intended to be representative; however, certain trends are evident from Figure 22. One is that methods 4 and 7 perform slightly better than other methods requiring comparable time. Method 5, which uses the recoded multiplier with 1's complement arithmetic, does not show up well due to the fact that the additional circuitry for high-speed subtraction is included in the complexity index for this method. A slow method of subtracting, such as the complementing and adding scheme discussed in Chapter V, is not satisfactory here. If the high-speed subtract logic is already built into the computer, then Method 5 becomes much more attractive. In any case, if much faster multiplication is desired, the recoding-of-the-multiplier method must be used. The slope of the straight lines in Figure 22 indicates that the complexity doubles with every decrease of 6.2 μsec. in multiply time.

Figure 22 also shows quite clearly the decided advantage in time that the 2's complement methods have over the 1's complement methods. This advantage in time must be weighed against the difficulties involved in implementing the arithmetic section for a computer using the 2's complement system.

9.8 Selection of a Multiplication Method

A computer designer could use the information in a curve such as Figure 22 to determine which type of multiplication was best suited to the particular problem at hand. If most of the other instructions for the computer had been already specified, then the designer would know for a particular problem having a certain number of multiplications to be performed what the maximum acceptable multiply time would be. If it were desirable to solve that one problem at the smallest cost, then the least complex multiply logic having an acceptable multiplication time would be the proper choice. If, however, some spare running time could be utilized effectively, then a curve such as Figure 22 would indicate the trade-off between extra running time and additional complexity. The curve might be used also to evaluate possible special-purpose commands other than multiplication. Suppose, for example, there are two possible methods of saving a certain amount of running time, a special purpose order and a faster multiplication. Figure 22 and the information as to the additional complexity required for implementing the special-purpose order could be used to decide which was the more economical method of realizing the savings in time.

It has already been seen that with the addition of a few instructions, the TX-O could solve the F-100A problem at a sufficiently rapid rate using a full add and shift multiplier such as method 1. If additional spare time were desired, then method 4 would save approximately 1.8 msec. of worst-case time. But, according to Figure 22 the use of method 4 would increase the complexity of the multiply logic over that required by method 1 by about 60%.

It should be pointed out that the whole discussion in this chapter has been carried out for maximum multiply time. It is difficult to believe that all the multiplications for a given solution cycle of a complex problem could possibly require the maximum amount of time. Therefore, when the time required for many multiplications is desired, it would probably be safe to use some value less than the maximum multiply time.

CHAPTER X

SUMMARY AND CONCLUSIONS

10.1 Introduction

The plan of attack followed here in outlining the functional design of a Special-Purpose Digital Computer for Real-Time Flight Simulation was to first determine in exact detail the nature of the problem for a specific case, (the F-100A) then to write a complete program for this model largely employing the order code of an existing computer, and lastly, on the basis of an analysis of this program to outline a new functional computer design that would be suitable for a wide range of real-time simulation problems. Where possible, the major design alternatives have been put on a quantitative basis, that is, in terms of such factors as complexity, memory size, and running time. From the variety of individual areas investigated in this study, the general features of a practical simulation computer begin to emerge. In this chapter, an attempt will be made to summarize the individual results and put them in proper perspective relative to the overall design problem. These results fall into two general areas, the analysis of the problem and the functional design of the computer.

10.2 Results of the Analysis of the Problem

The areas that were investigated pertaining to the analysis of the problem were aerodynamic function generation, the use of subroutines, and integration formulas. A brief summary of the results in each of these areas is given below.

Aerodynamic Function Generation

Comparisons between the UDOFT and TX-O methods of function generation indicate: 1) The TX-O method of using the aircraft manufacturer's data in its original form is superior to the UDOFT method in accuracy for the number of breakpoints used by the respective studies. 2) The TX-O method is superior in convenience of setting up and changing non-linear functions. 3) The TX-O method is comparable to the UDOFT method in data storage

requirements when the number of breakpoints used for the TX-O method is reduced to make the accuracy comparable to that of UDOFT. And 4) The TX-O method is a little slower in running time than the UDOFT method, the difference being approximately 10% of the total time consumed by function generation. This penalty in running time is independent of the number of breakpoints used for the TX-O method, so that the method would still be only about 10% slower even when the accuracy is made superior to the UDOFT accuracy.

No corresponding analysis has been made of the method of handling non-linear functions in the engine, hydraulic system, or instruments. From the results of the analysis of aerodynamic functions, however, it is felt that an analysis of these other areas would yield similar results. The recommendation therefore, is to handle all non-linear functions by the TX-O method, that is by linear interpolation between discrete points.

Subroutines

In general, the use of subroutines decreases the memory requirements of a given program at the expense of running time. Since memory costs money, the use of subroutines constitutes just another method of trading running time for money. Therefore, the extent to which subroutines are employed must be decided in the context of the trade-off obtainable by other means, such as the incorporation of additional general-purpose orders, special-purpose orders, and faster multiplication. For a given machine and problem, the subroutine decision will be determined by whether one has a surplus of running time or memory capacity available.

No definite conclusions on the use of subroutines have yet been reached for running the F-100A problem on the TX-O. Subroutines were not used in the preliminary TX-O program, and, in fact, their use could not even be considered unless some means were found to decrease the running time below the maximum acceptable running time of 50 msec. If orders such as those suggested in Chapter V were available, then subroutines could be used to advantage to decrease memory requirements. For example, in using subroutines for function generation, the trade-off ratio between memory and running time lies roughly in the range 46 to 220 words saved per extra one msec. of running time (present TX-O order code). When the order code is augumented by the instructions recommended in this study, the trade-off ratio is between 20 and 160.

Integration Formula

Integration formulas were not directly investigated in this study, but the results of an empirical analysis carried out at the M.I.T. Electronic Systems Laboratory in 1958 have been utilized. They show that a trapezoidal formula at 25 solutions per second is comparable in accuracy to UDOFT's Mod Gurk at 40 solutions per second and is more accurate in reproducing violent transients than Mod Gurk at 20 solutions per second. On the basis of this study, trapezoidal integration was chosen for the TX-O program.

10.3 Specification of Computer Characteristics

The specification of the important characteristics of a computer well suited to a particular real-time simulation problem was the other main area of research of this study. The main computer characteristics considered are: number representation, order code, and word length. These areas are discussed briefly below.

Number Representation

The TX-O uses the 1's complement representation of negative numbers, and it would not be reasonable to suggest changing it to some other system. The 1's complement does have many advantages. For example, only add logic is required to handle both positive and negative numbers and the complement of a number can be formed easily. The 2's complement system shares some, but not all, of the advantages of the 1's complement system. The 2's complement system does have the very important advantage of a faster multiplication (2.9 μsec. faster in the multiply circuits analyzed in Chapter IV). In the design of a computer for a specific problem, the importance of multiplication time would govern whether the 2's complement system was worth the additional complexity that its implementation entails.

Order Code

For the F-100A problem, which is representative of a wide class of real-time simulation problems, the TX-O order code, with a few changes, was found to be quite satisfactory. The additions to the TX-O order code recommended for the F-100A problem and their proposed operation times are:

1. Addressable multiply order (25 μsec.)
2. Non-addressable divide order (40 μsec.)
3. Accumulator right shift k places (12 μsec.)

4. Accumulator left shift k places (12 μsec.)
5. Special level-select order (variable)
6. Load accumulator order (12 μsec.)
7. Indexable load accumulator order (12 μsec.)
8. Subtract order (12 μsec.)
9. Indexable subtract order (12 μsec.)

The incorporation of these orders would reduce the worst-case running time to 42.725 msec. and the memory requirement to 6913 registers.

The cheapest acceptable multiply for the TX-O would be a full-add-and-shift multiply similar to the 25 μsec. addressable multiply on the Digital Equipment Corporation's PDP-1. Multiply logics offering maximum execute times as low as 11.65 μsec. were investigated in this study, but the complexity of these faster schemes is substantially greater.

In the area of high-speed multiplication, an investigation of a method to yield a correct double-length product using a 1's complement machine with the recoded multiplier technique would be quite promising, if the implementation could be accomplished without the use of an additional register. If a simple method were found, then it would be possible to combine some of the advantages of the recoded multiplier technique with the advantages of the 1's complement number system to yield a very good multiply method.

Various other orders have been suggested and, although they were not considered useful enough for the F-100A problem to warrant their inclusion in the TX-O order code, they might prove more worthwhile for other problems. The design method used throughout this study has been to re-program the F-100A problem using suggested modifications in order to determine the specific savings in time and memory that result. This data and a knowledge of the cost of implementating the modification provide a means of comparison with other suggested modifications.

Word Length

The word length requirement is affected by two different factors:

1. Dynamic range of variables.
2. Number of instructions and the size of the directly-addressable memory.

Of these factors, the affect of integration on the dynamic range of variables, particularly altitude, was found to impose the most severe requirement on word length. However, certain computations can be performed by double-

precision methods, at the cost of both time and additional memory. For the F-100A problem, an 18 bit word length is felt to be just barely acceptable. Connelly[4] concludes that a word length of 24 bits would be a good word length for the simulation of modern fighter aircraft. These 24 bits include additional tag bits that provide more versatile control of the analog-digital conversion operations. This latter feature would be helpful for problems where the analog input and output are more time consuming than for the F-100A. Since the cost of a digital simulator is the critical problem at the present time and since cost is roughly proportional to word length, accepting the inconvenience and added running time of the 18-bit word length is indicated for the special-purpose OFT computer.

Other Miscellaneous Characteristics

The use of floating point arithmetic was investigated by Connelly.[4] His conclusions were that when variables approached their maximum values, the dynamic range considerations imposed by integration would have the same effect on the word length of the characteristic part of the floating-point representation as on fixed point numbers. Therefore, when the number of bits required by the exponent is added to the number of bits required for the characteristic, the result is an unusually long word length. Floating point arithmetic would simplify programming, since scaling operations would be performed automatically by the computer. For a general-purpose simulation facility, the additional complexity and word length of floating-point implementation might prove worthwhile.

One other possibility that has not been investigated in this study is the use of two independent memories to increase the average instruction rate while using circuitry of the same basic speed. There are many possibilities, such as the UDOFT scheme of using separate memories for instructions and data or the use of separate memories for even and odd numbered memory locations. Both of these methods would result in faster instruction access when the program was not branching. The use of multiple memories, if designed to place no programming restrictions on the user and to impose no excessive time penalty for branching, could result in a substantial savings in time at a rather modest cost in equipment.

10.4 Conclusion

The main question investigated in this study was the ability of a TX-O class computer to solve the complete F-100A problem in real-time. This was

necessary in order to establish the relative capabilities of analog and digital techniques for real-time simulation. For the study to be of practical significance, the machines compared must also be economically competitive.

With the inclusion of a 25 μsec. addressable multiply order, a 40 μsec. divide order, and the shift right and shift left k places orders, and employing a memory size of 7463 words, the TX-O would be able to solve the F-100A problem at 20 solutions per second.

The ability of the TX-O to solve more complex real-time simulation problems would be further enhanced by the inclusion of a load accumulator order, a subtract order, and a special level-select order. Whereas the emphasis in this study has been on the specific requirements of the F-100A problem, the design methodology and the various design trade-offs prescribed should be applicable to the functional analysis of other simulation problems of greater or lesser complexity.

BIBLIOGRAPHY

1. Kennedy, O; Moroney, R; and Morse, M., An Experimental Analog-Digital Flight Simulator, M. I. T. Servomechanisms Laboratory, Rpt. 45-2, January, 1959. (Also published by Naval Training Device Center, Port Washington, New York as NAVTRADEVCEN 45-2.)

2. UDOFT Simulation Program, Final Report FR77-1N, Sylvania Electronic Systems, Needham, Mass. May 1960.

3. Simulation of a Supersonic Fighter Using a Digital Computer Moore School Rpt. 55-20, University of Pennsylvania, Philadelphia May, 1955.

4. Connelly, M. E., Analog-Digital Computers for Real-Time Simulation, M. I. T. Electronic Systems Laboratory, Final Rpt. ESL-FR-110, June, 1961. (Also published by Naval Training Device Center, Port Washington, New York as NAVTRADEVCEN 594-1).

5. Connelly, M. E., Simulation of Aircraft, M. I. T. Servomechanisms Laboratory, Rpt. 7591-R-1, February, 1958. (Also published by Naval Training Device Center Port Washington, New York as NAVTRADEVCEN 7591-R-1.)

6. Flight Trainer Digital Computer Study, Moore School Rpt. 51-28, University of Pennsylvania, Philadelphia, March, 1951.

7. Dwyer, P. S., Linear Computations, John Wiley, New York, 1951.

8. A More Powerful Order Code for the TX-O, Memorandum M-5001-22, Department of Electrical Engineering, M. I. T., Cambridge 39, Mass. May, 1960.

9. Aerodynamic Data for the Design of the F-100-A Simulator North American Aviation Rpt. NA53-592, North American Aviation, Los Angles 45, California.

10. Gurk, H. M., and Rubinoff, M., Numerical Solutions of Differential Equations, Proceedings of the 1954 Eastern Joint Computer Conference, December, 1954.

11. Aero Equations for the F-100A, Sylvania Electronic Systems, Needham, Mass.

12. Programming Manual for the UDOFT Computer, Sylvania Electronic Systems, Needham, Mass. August, 1959.

13. The Bendix G-20 Central Processor Machine Language, T 23-2, Bendix Computer Division, Los Angeles 45, Calif. January, 1961.

BIBLIOGRAPHY (continued)

14. Programmed Data Processor - 1, F-15A, Digital Equipment Corporation, Maynard, Mass., March, 1961.

15. 160 Computer Programming Manual, Publication No. 023a, Control Data Corporation, Minneapolis 15, Minn., 1960.

16. Lyon, E.F., A Study of Arithmetic Elements for the TX-0, M.S. Thesis, Department of Electrical Engineering, M.I.T., Cambridge 39, Mass., May 1959.

17. On the Design of a Very High-Speed Computer, Report No. 80, University of Illinois Graduate College Digital Computer Laboratory, Urbana, Illinois, October, 1957.

18. Methods of High-Speed Addition and Multiplication, NBS Circular 591, National Bureau of Standards, Washington, D.C., February, 1958.

19. Ledley, R. S., Digital Computer and Control Engineering, McGraw-Hill Book Company, New York, 1960.

20. Salter, F., High-Speed Transistorized Adder for a Digital Computer, IRE Transactions on Electronic Computers, Vol. EC-9, pp. 461-464, December, 1960.

21. Hendrickson, H.C., Fast High-Accuracy Binary Parallel Addition, IRE Transactions on Electronic Computers, Vol. EC-9, pp. 465-469, December 1960.

22. Binsack, J., A Pulsed-Analog and Digital Computer for Function Generation, Electronic Systems Laboratory, Scientific Report 8494-R-2, M.I.T., Cambridge 39, Mass, October, 1960 (AFCRL TN 60-1111)

23. Gilmore, J., A Functional Description of the TX-0 Computer, Memo 6M-4789, Division 6, M.I.T., Lincoln Laboratory, Lexington, Mass.

24. Hill, J., A Fast Digital-Analog Converter and Its Accuracy Limitations, M.S. Thesis, Department of Electrical Engineering, Electronic Systems Laboratory, M.I.T., Cambridge 39, Mass., November, 1959.

25. The Future TX-0 Instruction Code, Memorandum M-5001-29, Department of Electrical Engineering, M.I.T., Cambridge 39, Mass., Sept. 1960.

26. Programming for the TX-0, Memorandum M-5001-13-1, Department of Electrical Engineering M.I.T., Cambridge 39, Mass, October, 1960.

27. A Symbolic Utility Program for TX-0, Memorandum M-5001-23, Department of Electrical Engineering, M.I.T., Cambridge 39, Mass., July 1960.

28. Marco IIA, Memorandum M-5001-5-1, Department of Electrical Engineering, M.I.T., Cambridge 30, Mass., April, 1961.

APPENDIX I

A BRIEF DESCRIPTION OF THE TX-O

The TX-O[23] is an experimental digital computer utilizing transistor circuitry; it was designed and built by the M.I.T. Lincoln Laboratories to test out a 65,536-word core memory. In 1958, it was turned over to the Electrical Engineering Department at M.I.T. with a smaller, transistor-driven 4096-word core memory (18 bits), which has since been expanded by an additional 4096 words. The memory cycle time is 6 μsec., a complete operation taking two memory cycles or 12 μsec. The instruction is obtained in cycle zero and the operand in cycle one. Each memory cycle has eight time pulses.

The TX-O input facilities include a photoelectric paper tape reader, eighteen toggle switch storage registers, a toggle switch accumulator and buffer register, an on-line Flexowriter, and a light pen used in conjunction with an oscilloscope output display. The light pen permits logical decisions based on the manual selection of points plotted on the oscilloscope. An Epsco Datrac encoder (model B-611) can be used to convert analog voltages within the ranges of ±1.000, ±10.00, or ± 100.0 volts into 11-bit digital numbers. The unit converts at a rate of 2 μsec. per bit. Pulsing the encoder activates a sample and hold circuit at the input. Approximately 22 μsec. later, a digital output of 11 parallel bits is available. Another TX-O pulse, under program control, will strobe this number into the TX-O live register.

The output facilities consist of the aforementioned oscilloscope (511 x 511 grid) with a Polaroid camera attachment, a Flexowriter tape punch and printer and the usual console indicator displays. The digital-to-analog decoder used on the TX-O was designed by Hill.[24] Fifteen parallel transistor switches excite a ladder network of precision registers; a complete conversion is performed within one microsecond, the digital data being strobed into the decoder register from the TX-O live register.

The operation code of the TX-O[25] consists of nineteen orders of which eighteen are addressable. The nineteenth order, operate, does not refer to a memory location, but is a microprogramming instruction which specifies a logical or arithmetic operation to be carried out within the arithmetic registers of the machine. Within a single operate order, a programmer may build up a wide variety of operation sequences by selecting the appropriate combinations of individual operations.

The eighteen addressable instructions are:

Instruction	Mnemonic	Operation Performed
store accumulator	sto y	replace c(y) with c(ac)
store accumulator indexed	stx y	replace c(y+c(xr)) with c(ac)
store index register in address	sxa y	replace the address portion of c(y) with c(xr)
add one to memory	ado y	add one to c(y) and leave the result in c(y) and c(ac)
store live register	slr y	replace c(y) with c(lr)
store live register indexed	slx y	replace c(y+c(xr)) with c(lr)
add to accumulator	add y	add c(y) to c(ac)
add to accumulator indexed	adx y	add c(y+c(xr)) to c(ac)
load index register	ldx y	replace c(xr) with the sign and address portion of c(y)
augument index register	aux y	add the sign and address portion of c(y) to c(xr)
load live register	llr y	replace c(lr) with c(y)
load live register indexed	llx y	replace c(lr) with c(y+c(xr))
transfer on negative accumulator	trn y	take the next instruction from y if c(ac) is negative
transfer on zero accumulator	tze y	take the next instruction from y if c(ac) is zero
transfer and set index register	tsx y	replace c(xr) with c(pc) + 1 and take the next instruction from y

Instruction	Mnemonic	Operation Performed
transfer and index	tix y	if c(xr) is not zero, take the next instruction from y and decrease magnitude of c(xr) by one - if c(xr) is zero, take the next sequential instruction
unconditional transfer	tra y	take the next instruction from y
unconditional transfer indexed	trx y	take the next instruction from y+c(xr)

All addressable instructions require 12 μsec. with the exception of the transfers, which require 6 μsec. when they transfer, and the add one (ado y) and store index in address (sxa y) instructions, which require 18 μsec.

The micro-orders that can be used with the operate order are:

Cycle No.	Time Pulse	Mnemonic	Operation Performed
0	7	amb	replace c(mbr) with c(ac)
0	8	cla	clear c(ac) to plus zero
The Memory Buffer Register is Always Cleared at the Beginning of Cycle One			
1	2	xmb	replace c(mbr) with c(xr)
1	2	com	complement the accumulator
1	3	anb	the logical product (and) of c(lr) and c(mbr) replaces c(mbr)
1	3	orb	the logical sum (inclusive or) of the c(lr) and c(mbr) replaces the c(mbr)
1	4	mbl	replace c(lr) with c(mbr)
1	4	lmb	replace c(mbr) with c(lr)
1	5	pad	the exclusive or between the c(ac) and c(mbr) replaces c(ac)

Cycle No.	Time Pulse	Mnemonic	Operation Performed
1	6	shr	shift the accumulator right one position leaving the sign bit unchanged
1	6	cyr	cycle the accumulator right one position replacing bit 0 with bit 17
1	7	cry	a partial add (pad) followed by a carry operation (cry) will replace c(ac) with the result of a full add between c(ac) and c(mbr)
1	8	mbx	replace c(xr) with c(mbr)

With certain restrictions, groups of these instructions may be combined into a single operate instruction that will still require only 12 μsec. Input-Output instructions, which occur after time pulse 8 of cycle 0, are also part of the operate class group and can be combined with the micro-orders listed above.

Additional information about the TX-0 and the assembly and communications facilities available can be found in References 26 to 28.

In several places in this report, the one's complement system of binary numbers has been mentioned. Many computers, including the TX-0, use this system because the arithmetic logic for addition may also be used for subtraction. The negative of a positive binary number is very simply formed in the one's complement system by interchanging zeros and ones. One somewhat confusing aspect of the convention, however, is that there are two versions of zero, one positive and one negative.

+ zero = 000000000000000000

- zero = 111111111111111111