GEMINI SPACECRAFT
DIGITAL COMPUTER ARCHITECTURE

revision 1.0

John Pultorak
January 2, 2005

Abstract
This report describes the architecture of the Gemini Spacecraft Digital Computer (from an external perspective), and is probably sufficient to construct a primitive emulator. The computer is described at the register-transfer level. The actual implementation was serial, but there is no attempt here to capture the serial operations.

This information was extracted from:

"NASA Project Gemini Familiarization Manual, Rendezvous and Docking Configurations" SEDR 300, Vol. 2, suppl, July 1, 1966.

This document is publicly available and was purchased from NASA CASI/STI (79N76135) by the author of this report. The cost, at the time, was around $35. It may be downloadable now.

Conventions
IBM assigns number 1 to the high-order (most significant) bit.

INSTRUCTION SET

Opcodes are in octal.

nmem opcode description

HOP 00 Fields in the memory location referenced by the operand (see HOP word) are used to change the next instruction address.

01	02	03	04	05	06	07	08	09	10	11	12	13
A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**0**	**0**	**0**	**0**

DIV 01 Divide, Contents of the memory location referenced by the operand are divided by the accumulator. The 24-bit quotient is available during the 5th word time following DIV. To obtain the quotient, make SPQ the fifth instruction following DIV. Four other instructions can placed between DIV and SPQ.

01	02	03	04	05	06	07	08	09	10	11	12	13
A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**0**	**0**	**0**	**1**

PRO 02 Process input or output. The input or output specified by the operand address is read into, or loaded from, the accumulator. For output, the accumulator is cleared if operand A9=1.

01	02	03	04	05	06	07	08	09	10	11	12	13
A1	**A2**	**A3**	**A4**	**A5**	**A6**	**–**	**--**	**A9**	**0**	**0**	**1**	**0**

RSU 03 Reverse subtract. The accumulator is subtracted from the memory location referenced by the operand. The result is placed in the accumulator.

01	02	03	04	05	06	07	08	09	10	11	12	13
A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**0**	**0**	**1**	**1**

ADD 04 Add. The contents of the memory location referenced by the operand is added to the contents of the accumulator. The result is placed in the accumulator.

01	02	03	04	05	06	07	08	09	10	11	12	13
A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**0**	**1**	**0**	**0**

SUB 05 Subtract. The contents of the memory location referenced by the operand is subtracted from the accumulator. The results is placed in the accumulator.

01	02	03	04	05	06	07	08	09	10	11	12	13
A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**0**	**1**	**0**	**1**

CLA 06 Clear and add. The accumulator is loaded with the contents of the memory location referenced by the operand.

01	02	03	04	05	06	07	08	09	10	11	12	13
A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**0**	**1**	**1**	**0**

AND 07 Bitwise AND. The contents of the memory location referenced by the operand are logically ANDed, bit-by-bit, with the accumulator. The result is placed in the accumulator.

01	02	03	04	05	06	07	08	09	10	11	12	13
A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**0**	**1**	**1**	**1**

MPY 10 Multiply. The contents of the memory location referenced by the operand are multiplied by the accumulator. The 24 high-order bits of the multiplier and multiplicand form a 26-bit product available during the second word time following MPY. To obtain the product, make SPQ the second instruction following MPY. One other instruction can placed between MPY and SPQ.

```
01  02  03  04  05  06  07  08  09  10  11  12  13
A1  A2  A3  A4  A5  A6  A7  A8  A9   1   0   0   0
```

TRA 11 Transfer. Transfers execution to the address specified in the operand field. Syllable and sector are unchanged. If A9 is 1, execution will occur in the residual sector.

```
01  02  03  04  05  06  07  08  09  10  11  12  13
A1  A2  A3  A4  A5  A6  A7  A8  A9   1   0   0   1
```

SHF 12 Shift. Shifts the contents of the accumulator left or right one or two places, as specified by the operand. An invalid code in the operand clears the accumulator. Left-shift enters zeroes in the low order positions. Right-shift copies the sign bit in the high-order positions. Valid codes:

	A1-A3	A4-A6
shift left 1 bit	ignored	3
shift left 2 bits	ignored	4
shift right 1 bit	1	2
shift right 2 bits	0	2

```
01  02  03  04  05  06  07  08  09  10  11  12  13
A1  A2  A3  A4  A5  A6   –  --   –   1   0   1   0
```

TMI 13 Transfer on minus accumulator sign. If the sign bit in the accumulator is negative, execution transfers to the address specified by the operand. Syllable and sector are unchanged. If A9 is 1, execution will occur in the residual sector.

```
01  02  03  04  05  06  07  08  09  10  11  12  13
A1  A2  A3  A4  A5  A6  A7  A8  A9   1   0   1   1
```

STO 14 Store. The accumulator is stored in the memory location referenced by the operand. The accumulator is unchanged.

```
01  02  03  04  05  06  07  08  09  10  11  12  13
A1  A2  A3  A4  A5  A6  A7  A8  A9   1   1   0   0
```

SPQ 15 Store product or quotient. The product is available in the second word time following MPY. The quotient is available in the fifth word time following DIV. The product or quotient is stored in the memory location referenced by the operand.

```
01  02  03  04  05  06  07  08  09  10  11  12  13
A1  A2  A3  A4  A5  A6  A7  A8  A9   1   1   0   1
```

CLD 16 Clear and add discrete. The discrete input selected by the operand is read into all accumulator bit positions.

```
01  02  03  04  05  06  07  08  09  10  11  12  13
A1  A2  A3  A4  A5  A6   –  --   –   1   1   1   0
```

TNZ 17 Transfer on non-zero. If the contents of the accumulator are non-zero, execution transfers to the address specified by the operand. Syllable and sector are unchanged. If A9 is 1, execution will occur in the residual sector.

```
01  02  03  04  05  06  07  08  09  10  11  12  13
A1  A2  A3  A4  A5  A6  A7  A8  A9   1   1   1   1
```

MDIU

DESCRIPTION

The MDIU is the pilot interface to the computer. It consists of Manual Data Keyboard and a Manual Data Readout units.

The Manual Data Keyboard is a keypad containing 10 buttons:

ZERO, 1,2,3,4,5,6,7,8,9

The Manual Data Readout has 3 buttons:

READOUT, CLEAR, ENTER

The Manual Data Readout also has a 7 digit decimal display. The first 2 digits show an address, and the last 5 digits show data.

The pilot can enter or display data for up to 99 addresses. Each address identifies a type of data. A display of zero address and data indicates pilot error.

OPERATION

The pilot enters data by depressing keys on the numeric keypad. The address is entered first (2 button presses). Then the data is entered (5 button presses). Each inserted digit is displayed for verification. After the data is entered, the pilot presses ENTER to store the data in the selected memory location.

The pilot checks data at an address by entering the 2 digit (2 button presses) address, and then pressing READOUT. The selected data is displayed.

The CLEAR button must be pushed before the first quantity of data is entered or displayed. If the pilot presses the CLEAR pushbutton during address or data entry, the previous entries are rejected, and the address and data must be reentered.

If the pilot enters an invalid address, enters more than 7 digits, or fails to insert an address before pressing ENTER or READOUT, the readout displays 7 zero digits indicating pilot error.

INTERFACES (CLD)

The following are accessed by the CLD instruction. The operand is bits (A1-A9) of the CLD instruction. 'x' means don't care.

		CLD operand (A1-A9)	
DI01	Data Ready	10x	A digit (0-9) key has been pressed.
DI02	Enter	20x	The ENTER key was pressed.
DI03	Readout	30x	The READ OUT key was pressed.
DI04	Clear	40x	The CLEAR key was pressed.

INTERFACES (PRO)

The following are accessed by the PRO instruction. The operand is bits (A1-A9) of the PRO instruction. 'x' means don't care.

		PRO operand (A1-A9)
DO30	Digit magnitude weight 1	03x
DO31	Digit magnitude weight 2	13x
DO32	Digit magnitude weight 4	23x
DO33	Digit magnitude weight 8	33x
DO40	Reset DI01, DI02, DI03	04x
DO41	Display device drive	14x
DO50	Digit select weight 1	05x
DO51	Digit select weight 2	15x
DO52	Digit select weight 4	25x
DI43	Read MDIU insert data	34x

PROTOCOL

When the pilot presses CLEAR, DI04 is set. The program polls this input, using the CLD instruction. When the program detects DI04, it sets DO40 off. This resets DI01, DI02, and DI03, and clears the MDIU keyboard input buffer. The program then sets DO41 off to reset the display drivers.

When the pilot presses a digit pushbutton, the BCD code is inserted into the keyboard input buffer and DI01 is turned on. The program reads the buffer into the Accumulator bits A1-A4 with DI43. After that, the program sets DO40 off.

To display the entered digit, the program uses DO50, DO51, and DO52 to select the digit to be displayed. Then, the program sets DO41 to turn on the display drivers. It then sends the BCD digit to the buffer using DO30, DO31, DO32, and DO33. The program waits 0.5 seconds (for the mechanical digit display to respond), and then sets DO40 and DO41 off.

After all seven digits are entered, the pilot depresses ENTER, which results in DI02 being set on. The program then sets DO40 off, converts the 5 decimal digits to binary, and stores them in the selected memory location according to the 2-digit address.

To read data out of the computer, the pilot enters the 2-digit address and depresses the READ OUT button. This causes DIO3 to be set on. The computer sets DO40 off, converts the requested quantity to BCD, and sends the BCD data to the display buffer one digit at a time in 0.5 second intervals.

The on/off state of the latches is controlled by the sign bit in the Accumulator. A positive sign bit sets the latch. A negative sign resets the latch.

MEMORY

Memory is 4096 words. Each word is 39 bits, consisting of three 13-bit syllables. IBM assigns number 1 to the leftmost bit.

```
      -------------- SYLLABLE --------------
      01 02 03 04 05 06 07 08 09 10 11 12 13    (bit position)


      --------------- WORD -----------------
       SYLLABLE 0   SYLLABLE 1   SYLLABLE 2
      01  ...   13 14  ...   26 27  ...   39    (bit position)


        -------------39 bit word -----------
          01      13   14      26   27      39
sector    SYLLABLE 0   SYLLABLE 1   SYLLABLE 2   8-bit address    12-bit
address
0     |             (256 words)              | 000 - 377    0000 - 0377
1     |             (256 words)              | 000 - 377    0400 - 0777
2     |             (256 words)              | 000 - 377    1000 - 1377
3     |             (256 words)              | 000 - 377    1400 - 1777
4     |             (256 words)              | 000 - 377    2000 - 2377
5     |             (256 words)              | 000 - 377    2400 - 2777
6     |             (256 words)              | 000 - 377    3000 - 3377
7     |             (256 words)              | 000 - 377    3400 - 3777
10    |             (256 words)              | 000 - 377    4000 - 4377
11    |             (256 words)              | 000 - 377    4400 - 4777
12    |             (256 words)              | 000 - 377    5000 - 5377
13    |             (256 words)              | 000 - 377    5400 - 5777
14    |             (256 words)              | 000 - 377    6000 - 6377
15    |             (256 words)              | 000 - 377    6400 - 6777
16    |             (256 words)              | 000 - 377    7000 - 7377
17    |  (256 words; the "residual sector")  | 000 - 377    7400 - 7777
```

WORD REPRESENTATION

Instruction word

Instruction words are 1 syllable long. Instruction words can be read from any syllable of memory.

```
01  02  03  04  05  06  07  08  09  10  11  12  13    (bit position)
A1  A2  A3  A4  A5  A6  A7  A8  A9  OP1 OP2 OP3 OP4   (fields)
```

A1-A8 is the operand field. The low-order bit is A8.

A9 is the "residual" bit.

```
A9=1  Use memory sector 17 (the "residual" sector)
A9=0  Use sector register to reference memory.
```

OP1-OP4 is the opcode field. The low-order bit is OP4.

Data word

Data words are 2 syllables long. Numbers are 2's compliment, with 25 magnitude bits and 1 sign bit. The low-order bit is M25. The high-order bit is M1. 'S' is the sign.

Data words are read from syllables 0 and 1 of memory (normal mode) or syllable 2 only if the "special syllable bit" is set in the HOP word.

```
01  02  03  04  05  06  07  08  09  10  11  12  13    (bit position)
M25 M24 M23 M22 M21 M20 M19 M18 M17 M16 M15 M14 M13   (bit code)

14  15  16  17  18  19  20  21  22  23  24  25  26    (bit position)
M12 M11 M10 M9  M8  M7  M6  M5  M4  M3  M2  M1  S     (bit code)
```

The least significant bit (M25) comes first because arithmetic operations are done serially, beginning with the low order bits.

HOP word

The HOP word transfers execution to a different sector in memory.

```
01  02  03  04  05  06  07  08  09  10  11  12  13    (bit position)
A1  A2  A3  A4  A5  A6  A7  A8  A9  S1  S2  S3  S4    (fields)

14  15  16  17  18  19  20  21  22  23  24  25  26    (bit position)
--  SYA SYB --  S5  --  --  --  --  --  --  --  --    (fields)
```

A1-A8 identifies the address of the next instruction to execute in the new sector.

A9 is the "residual" bit.

```
A9=0  Use sector bits S1-S4 to select the new sector.
A9=1  Use memory sector 17 (the "residual" sector) for the new sector.
```

S1-S4 identifies the new sector. Overridden if A9=1.

SYA and SYB select the new syllable.

```
SYB   SYA   selected syllable
```

0	0	0
0	1	1
1	0	2

S5 (special syllable bit) sets the mode for reading data words.

S5=0 Read data words from syllables 0 and 1 (normal mode).

S5=1 Read data words from syllable 2 only.

REGISTERS

This is a serial computer using early technology. Many registers are implemented by glass delay lines: data is maintained by regenerating it and recirculating it through the delay line. These representations are functional:

OPR Operation Register (4 bit)

```
01  02  03  04   (bit position)
OP1 OP2 OP3 OP4
```

SCR Sector Register (4 bit)

```
01  02  03  04   (bit position)
S1  S2  S3  S4
```

IAR Instruction Address Register (9 bit)

```
01  02  03  04  05  06  07  08  09   (bit position)
A1  A2  A3  A4  A5  A6  A7  A8  A9
```

A1-A8 identifies the "word position" of the next instruction to execute in the new sector.

A9 is the "residual" bit.
A9=0 Use sector bits S1-S4 to select the new sector.
A9=1 Use memory sector 17 (the "residual" sector) for the new sector.

SYR Syllable Register (2 bit)

```
01  02   (bit position)
SYB SYA
```

SYB	SYA	selected syllable
0	0	0
0	1	1
1	0	2

SSB Special Syllable Bit (1 bit)
SSB=0 Read data words from syllables 0 and 1 (normal mode).
SSB=1 Read data words from syllable 2 only.

MAR Memory Address Register (8 bit)

```
01  02  03  04  05  06  07  08   (bit position)
A1  A2  A3  A4  A5  A6  A7  A8
```

A1-A8 identifies a memory address inside a sector.

ACC Accumulator (26 bit)

```
01  02  03  04  05  06  07  08  09  10  11  12  13    (bit position)
M25 M24 M23 M22 M21 M20 M19 M18 M17 M16 M15 M14 M13   (bit code)

14  15  16  17  18  19  20  21  22  23  24  25  26    (bit position)
```

M12 M11 M10 M9 M8 M7 M6 M5 M4 M3 M2 M1 S (bit code)

The least significant bit (M25) comes first because arithmetic operations are done serially, beginning with the low order bits.

TIMING

The computer instruction cycle is 5 phases long: (PA, PB, PC, PD, PE).

All instructions execute in 1 cycle, except for MPY and DIV. MPY requires 3 cycles, and DIV requires 6 cycles.

A MPY or DIV instruction starts the operation in the multiply-divide element. The program must obtain the result (using the SPQ instruction) at the right time.

When MPY is commanded, the product is available two cycle times later. When DIV is commanded, the quotient is available five cycle times later.

It is possible to have one other instruction run concurrently between MPY and SPQ, and four other instructions run concurrently between the DIV and SPQ. However, an MPY or DIV is always followed by a SPQ before a new MPY or DIV is given.

Description of the instruction cycle:

PA Phase A: Read the 13-bit instruction word from memory and store it in the instruction address register (IAR), Store the 4-bit operation code in the operation register (OPR).

PB Phase B: Copy the operand address from the instruction address register (IAR) to the memory address register (MAR). Increment the instruction address in the instruction address register (IAR).

PC/D Phases C and D: Perform the instruction specified in the operation register (OPR).

PE Phase E: Transfer the address of the next instruction from the instruction Address Register (IAR) to the memory address register (MAR). This prepares the computer to read the next instruction in the upcoming Phase A.

REGISTER TRANSFERS

Disclaimer: For simplicity (and due to lack of information), operations in phases C and D have been lumped into one phase C/D category. A few operations SEDR 300 identifies for phases A, B, or E have also been moved to C/D. The register transfers herein are my interpretation, and have not been tested in an emulator. Doubtlessly, there are errors and omissions.

```
      KEY:
      []    specifies an address for a memory transfer
      -     specifies a range of bits
      :     (colon) terminates a control function
      *     logical AND
      +     logical OR
      '     logical invert (NOT)
      <-    denotes transfer of information
      ()    denotes a portion of a register
      plus  arithmetic addition operator
      minus arithmetic subtraction operator
      #     comment follows

initial:
            # Initialization: start execution in sector 0, address 0,
            # syllable 0,
      SCR <- 0
      MAR <- 0
      SYR <- 0


PA * IAR(A9)':
            # Phase A: if the residual bit is zero, construct 12-bit
            # effective address for next instruction from the sector
            # register and the memory address register.
      ea(1-4) <- SCR, ea(5-12) <- MAR


PA * IAR(A9):
            # Phase A: if the residual bit is one, construct 12-bit
            # effective address for next instruction from the residual
            # sector (17) and the memory address register.
      ea(1-4) <- 17, ea(5-12) <- MAR


PA:
            # Phase A: using the 12-bit effective address, fetch the
            # next instruction from the selected syllable.
      IAR <- MEM[ea, SYR]
      OPR <- MEM[ea, SYR]


PB:
            # Phase B: prepare to access the operand. Bump the address for the
            # next instruction.
      MAR <- IAR(A1-A8)
      IAR(A1-A8) <- IAR(A1-A8) plus 1


PC/D * IAR(A9)':
            # Phases C and D: construct 12-bit effective address for
```

```
            # the operand from the sector register and the memory
            # address register.
      ea(1-4) <- SCR, ea(5-12) <- MAR


PC/D * IAR(A9):
            # Phases C and D: construct 12-bit effective address for
            # the operand from the residual sector (17) and the
            # memory address register.
      ea(1-4) <- 17, ea(5-12) <- MAR


PC/D * SSB':
            # Phases C and D: using effective address, read operand
            #(normal mode).
      operand <- MEM[ea](S-M25) # read syllables 0,1


PC/D * SSB:
            # Phases C and D: using effective address, read operand
            # (single syllable mode).
      operand(S-M12) <- MEM[ea](S-M12) # read syllable 1 only
      operand(M13-M25) <- 0


PC/D * HOP:
            # Phases C and D: execute HOP instruction.
      IAR <- MEM[ea](A1-A9)
      SCR <- MEM[ea](S1-S4)
      SYR(SYB) <- MEM[ea](SYB)
      SYR(SYA) <- MEM[ea](SYB)
      SSB <- MEM[ea](S5)


PC/D * CLA:
            # Phases C and D: execute CLA instruction.
      ACC <- operand


PC/D * ADD:
            # Phases C and D: execute ADD instruction.
      ACC <- ACC plus operand


PC/D * SUB:
            # Phases C and D: execute SUB instruction.
      ACC <- ACC minus operand


PC/D * AND:
            # Phases C and D: execute AND instruction. Does a bitwise
            # logical and of the operand with the accumulator.
      ACC <- ACC * operand


PC/D * TRA:
            # Phases C and D: execute TRA instruction.
      IAR <- MAR(A1-A8)
```

```
PC/D * TMI * ACC(S):
            # Phases C and D: execute TMI instruction. Branch if
            # accumulator is minus.
      IAR <- MAR(A1-A8)


PC/D * TNZ * ACC:
            # Phases C and D: execute TNZ instruction. Branch if
            # accumulator is nonzero.
      IAR <- MAR(A1-A8)


PC/D * RSU:
            # Phases C and D: execute RSU instruction.
      ACC <- operand minus ACC


PC/D * SHF * MAR(A4-A6)=3:
            # Phases C and D: execute SHF instruction (shift left one place).
      ACC(S-M24) <- ACC(M1-M25)
      ACC(M25) <- 0


PC/D * SHF * MAR(A4-A6)=4:
            # Phases C and D: execute SHF instruction (shift left two places).
      ACC(S-M23) <- ACC(M2-M25)
      ACC(M24) <- 0
      ACC(M25) <- 0


PC/D * SHF * MAR(A1-A3)=1 * MAR(A4-A6)=2:
            # Phases C and D: execute SHF instruction (shift right one place).
      ACC(M2-M25) <- ACC(M1-M24)
      ACC(M1) <- ACC(S)


PC/D * SHF * MAR(A1-A3)=0 * MAR(A4-A6=2):
            # Phases C and D: execute SHF instruction
            # (shift right two places).
      ACC(M3-M25) <- ACC(M1-M23)
      ACC(M1) <- ACC(S)
      ACC(M2) <- ACC(S)


PC/D * STO:
            # Phases C and D: execute STO.
      MEM[ea](S-M25) <- ACC(S-M25)


PC/D * MPY:
            # Phases C and D: execute MPY. Send 24-bit inputs to
            # the multiply-divide element. Product (26-bit) is
            # available during PC/D of the third instruction cycle.
      multiplier <- ACC(S-23)
      multiplicand <- operand(S-23)


PC/D * DIV:
            # Phases C and D: execute DIV. Send 26-bit inputs to
            # the multiply-divide element. Quotient (24-bit) is
```

```
            # available during PC/D of the sixth instruction cycle.
      divisor <- ACC(S-M25)
      dividend <- operand(S-23)


PC/D * SPQ:
            # Phases C and D: execute SPQ. Store the product or
            # quotient produced by the multiply-divide element.
      MEM[ea](S-M25) <- result


PC/D * CLD:
            # Phases C and D: execute CLD. Copied to all bits in the ACC.
      ACC(S-M25) <- discrete input selected by operand.


PC/D * PRO * input * IAR(A9)':
            # Phases C and D: execute PRO. OR inputs with ACC if A9 is a 0.
      ACC <- ACC + input


PC/D * PRO * input * IAR(A9):
            # Phases C and D: execute PRO.
      ACC <- input


PC/D * PRO * output * IAR(A9)':
            # Phases C and D: execute PRO.
      output <- ACC


PC/D * PRO * output * IAR(A9):
            # Phases C and D: execute PRO. Clear ACC if A9 is a 1.
      output <- ACC
      ACC <- 0


PE:
            # Phase E: prepare to fetch the next instruction.
      MAR <- IAR(A1-A8)
```