

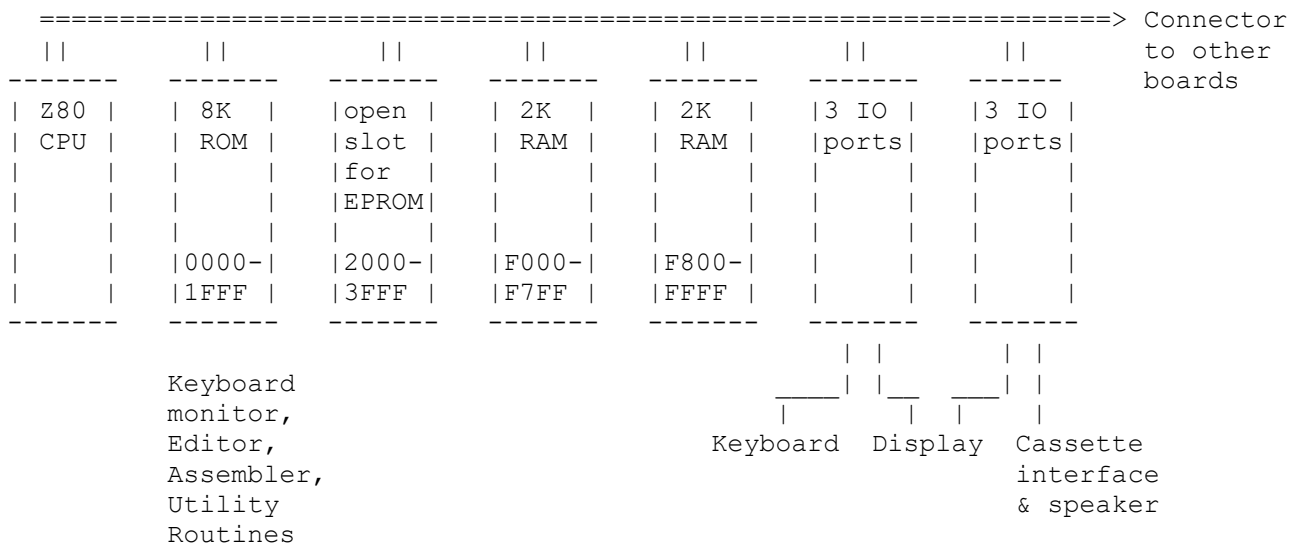
Need:

1. MPF-I's to sign out
2. Transparencies: Overall block-diagram of MPF-I  
Z80 Architecture  
Z80 Instruction set (4)  
Z80 Memory  
Z80 IO Timing  
Excerpt from Z80 Timing Table  
Z80 pinouts  
Z80 M1 cycle timing  
Z80 INT timing  
Z80 NMI timing  
Z80 BUSRQ timing  
MPF-I circuit diagrams (3)  
74LS138 diagram
3. Handouts: Overall block-diagram of MPF-I  
Monitor, editor, and assembler commands  
Z80 Architecture;  
Z80 Instruction set (4)
4. Copy of Z80 Microprocessor Programming and Interfacing

#### I. Overview of the MPF-I hardware

- A. The MPF-I is a one-board microcomputer, built around a Z80 8-bit microprocessor.
- B. Show one to the class. Each box will contain:
  1. MPF-I unit itself
  2. Power supply
  3. Three manuals: User guide, monitor source code listing, experiment manual. Of these, the user guide will be most helpful.
  4. Each team will be issued an MPF-I, for which they will be responsible. (You may take your manuals out of the lab overnight for study, but you are responsible for returning them.)
  5. We also have some additional boards that connect to the MPF-I. These must be shared:
    - a. Two printers
    - b. Three EPROM programmers
    - c. Two speech synthesizers (not used in any required labs, but available for optional use if desired.)
- C. The MPF-I consists of a Z80 CPU, a ROM containing a monitor, editor, and two assemblers (all in less than 8K!), 4K of RAM, a 48 character keyboard, and a 20-character display.
- D. Overall block-diagram of MPF-I. (TRANSPARENCY, HANDOUT)
  1. Note: we will look at detailed schematics later in the lecture.
  2. Show chips on board.

System bus (40 bits)



E. Note that Memory addresses are assigned as follows:

1. 0000-1FFF (hex) are the monitor, editor, assemblers etc.
2. 2000-3FFF (hex) are dedicated to slot U3 on the board, into which one can insert an EPROM containing a user-written program or data (as we shall do later.)
3. F000-FFFF (hex) are the on-board RAM. These addresses are important to note, because programs we write will be stored here.
4. The remaining addresses are available for off-board expansion - such as the dissassembler that is built into the printer board.

## II. Overview of the MPF-I built-in software

- A. When the MPF-I is powered up, control passes to the monitor at memory address 0. The monitor then displays the start up message MPF-I PLUS. Once you start entering commands, however, the monitor prompt becomes an upside down 7 followed by a carat (SHOW).
- B. You should think of the monitor as playing a role similar to the command language interpreters on the VAX. The upside-down 7 + carat prompt is analogous to the \$ prompt of DCL.
- C. When at monitor command level, you can enter one of the other packages such as the editor or assembler. The commands for so doing are summarized in the top box of the handout. NOTE WELL: These are control keys, not letter keys - e.g. to re-enter the monitor, type CONTROL-Q. (CONTROL-Q functions like CONTROL-C on the VAX). In addition to control-Q, there are two other control characters we want to note: Control-P toggles the printer on and off, and Control-G toggles the keyclick bell on and off. (SHOW).
- D. For our early work, we will primarily be using the editor and the two-pass assembler. Note that the former has two entry points: CONTROL-E to enter the editor for creating a new program, and CONTROL-R to re-enter it to edit an existing program. We will discuss the details of using these two facilities first, before going on to talk about other monitor commands.
  1. When you first enter the editor using CONTROL-E, it will ask you for the starting address of a memory region it can use as a text buffer. For now, pressing return to accept its default value will be the simplest solution.
  2. The editor has basically two modes of operation: command mode, in which you issue commands to move around in the text, make changes etc., and input mode, in which you type new text.
    - a. When you enter the editor to create a new program you are thrown

- into INPUT mode immediately.
- b. To enter the input mode otherwise, type I return.
- c. To leave input mode, enter a blank line.

3. GO OVER OTHER COMMANDS ON HANDOUT. NOTE THAT ALL CAN BE ENTERED AS FIRST LETTER.

4. To leave the editor, type Q in command mode.

5. To enter the assembler, type CONTROL-A. Like the editor, it will begin by prompting you for RAM addresses for the program it is to assemble, for its symbol table, and for the object code. For now, just press return in response to the prompts to accept the default.

6. The assembler makes two passes over the text. During pass 1, it reports errors. An error will be reported as a line number, a memory address where its code is going, and an error code - eg. 3. FC04 \*I\*. When you have noted the error, press return and the assembler will go on - possibly to find further errors!

7. You may use the editor G command to go to the offending line number in order to try to figure out your error.

E. When at monitor command level you also have immediately available to you ODT-like functions. These are summarized on the rest of the first page of the handout. NOTE that these commands are invoked by typing the letter specified - not by control characters. (Very confusing - you should probably write in CTRL before the Q, E, R, A, L, D, B, and C in the top box.) Also, note that while the commands in the first box are executed immediately upon pressing the specified control character, the remaining commands require you to press the letter then press return. (NOTE: SEE MPF-I USER'S MANUAL CHAPTER 4 FOR FURTHER INFO.)

1. The R command allows you to display or alter the registers.

- a. If you type R followed by a register name, the specified register is displayed. If it is an 8-bit register, its partner is displayed with it.
- b. If you just type R, the AF and BC pairs are displayed.
- c. To look at different registers, you can press the up or down arrow keys. The down-arrow steps through the registers (2 16-bit pairs at a time) in this order:

AF and BC  
DE and HL  
A'F' and B'C' (displayed as A.F. and B.C.)  
D'E' and H'L'  
IX and IY  
SP and PC  
PC and I

- d. To alter a register, one types R, followed by the register name, followed by a colon, followed by the new value (either 2 or 4 hex digits) followed by return.

2. The M command allows you to display or alter memory locations. (Of course, you can only examine existing locations, and you can only alter locations in RAM - F000 to FFFF.)

- a. To examine a location, type M followed by a 4-digit address and return. MPF-I displays 4 consecutive bytes starting at that address.
- b. As with registers, the uparrow and downarrow keys may be used to step forward or backward through the range of addresses.
- c. To alter a location, type M followed by its address followed by a colon and a new value. You can alter several consecutive bytes at once by entering new values (as pairs of hex digits) separated

- by spaces.
- d. M followed by an address followed by a . followed by a second address can be used to dump a range of memory to the printer. (Of course, the printer must be connected.)
3. The I command allows you to insert one or more bytes of data, with the rest of memory contents being shifted up to make room for it.
    - a. Type I, followed by the address AFTER WHICH the new data is to go, followed by one or more data bytes.
    - b. All bytes up to FE00 will be shifted up to make room for the newly inserted bytes. Of course, this means that some bytes will be shifted off the end of memory. (The address space above FE00 is reserved for monitor scratchpad work and is not altered) Note: FE00 is the default limit. You can specify a different limit if you wish.
  4. The D command allows you to delete a byte of data, with the rest of memory contents being shifted down to make room for it. A zero is filled into the vacated byte. Again, memory above FE00 is untouched unless you specify a different limit.
  5. The G command allows you to start program execution. Just entering G followed by return will start you at the current PC value; however, you can enter a different starting address after the G if you wish.
  6. The B command allows you to set a breakpoint at a specified address. Only one breakpoint may be active at a time.
  7. The S command allows you to single step a program. S by itself executes the instruction pointed to by the PC; or you can type S followed by an address.
  8. The W command allows you to write to cassette tape. You type W, followed by the start address of the memory range you wish to write, followed by a space, followed by the end address, followed by a space and a 4 character filename. Be sure the recorder is running before pressing return.
  9. The L command, followed by a filename, reads a file back from tape to the location in memory from which it was written.
  10. The J command causes MPF-I to calculate for you the relative address needed in a relative jump instruction at a specified address jumping to a specified absolute address.

### III. Z80 CPU architecture

- A. The Z80 was developed by a team of engineers who formerly worked for Intel, the company that developed the first microprocessor. At the time, Intel's most powerful chip was the 8080 (the chip on which many home computers - including all that run CP/M - were built). Intel had begun an internal effort to develop an improved 8-bit microprocessor. Evidently, the team that ultimately developed the Z80 wanted to go farther from the original 8080 design than did corporate management. The result was that they left Intel to form Zilog corporation and to develop their own chip. (Intel went on to release an improved version of the 8080 known as the 8085; but the changes were mostly in terms of external connections such as power supply requirements rather than in terms of internal architecture.)
- B. One of the design goals of the Z80 was that it should be able to run software written for the 8080. This causes some peculiarities in the instruction set, as we shall see.
- C. The Z80 is a pure 8-bit microprocessor, which means that its internal registers and data paths (as well as its external data bus) are 8 bits

wide. Since 16-bit integers are the minimal size for many applications, this means that many ordinary integer arithmetic operations must be done in two separate 8-bit steps. However, addresses are 16 bits long; therefore, the internal data paths for addressing are 16 bits wide. We shall see that provision is made at several points to join two 8-bit registers together to form a 16-bit pointer to a memory location. TRANSPARENCY OF STRUCTURE - GO OVER.

1. Note external connections: 8-bit data bus, 16-bit address bus, plus 13 control lines. Together, these form one PHYSICAL BUS and two LOGICAL BUSES: a memory bus and an IO bus. The two logical busses use the same data and address busses, but different control lines. The memory bus is used for all instruction and data fetches, and the IO bus by the IN and OUT instructions.
  2. We will discuss details of the bus structure later.
- D. The Z80 is also basically a one-accumulator machine, which means that for most arithmetic and logical operations the A register (the accumulator) contains one of the source operands and receives the result of the operation. Again, this contrasts with the VAX, which is a general register machines on which any of the 16 general registers can participate fully in arithmetic and logical operations.
- E. The 8080 has 8 8-bit registers and 2 16-bit registers which are carried over directly into the architecture of the Z80:
1. The 16-bit PC contains the address of the next instruction to be executed (as on the VAX); however, it is not a general register and cannot be used for any other purpose as the PC on the VAX can.
  2. The 16-bit SP is a stack pointer; again, it is not a general register and cannot be used for anything else.
  3. The 8-bit A register is the accumulator, and participates in most arithmetic and logical operations.
  4. The 8-bit F register plays a role similar to that of the PSW on the VAX. Actually, only 5 of its bits are used - the other 3 have undefined values. These bits serve as condition codes to reflect the results of various arithmetic and logical operations.
  5. The 8-bit registers H and L are seldom used as 8-bit registers. More often they are paired to form a 16-bit register that is used to point to a memory location. Many memory-reference instructions require that the address of the item to be fetched or stored be in HL. (This corresponds roughly to mode 6 - register deferred - addressing on the VAX - but only this one register can be used.)
  6. The 8-bit registers B,C,D and E provide temporary storage for intermediate results of operations, and can also be paired up (BC, DE) to form 16-bit registers that can be used as pointers to memory cells.
  7. In some contexts the F register can be paired with A to form a 16 bit AF register. This is primarily done when pushing registers on a stack - they are pushed 16 bits at a time, so A and F are pushed together. (Of course, they cannot be used together as a pointer to memory as the other pairs can!)
- F. The Z80 more than doubled the number of bits of registers of the 8080.
1. In place of a single set of registers A,B,C,D,E,F,H,L, the Z80 has two sets - one designated A', B' etc. However, only one set can be in use at any time. The only operations possible on the alternate set is an exchange operation which swaps the two register sets wholesale. The primary use of this feature is to allow operating system components

(such as interrupt handlers) to be able to use registers without destroying application-program data. (Switching register sets is much faster than pushing all the registers on the stack.)

2. The Z80 has two index registers - IX and IY - which allow for indexed addressing similar to VAX displacement modes A,C,E. (The so-called index mode on the VAX is different). There was no provision for this on the 8080.
3. The Z80 has an I register that is used to provide more sophisticated interrupt vectoring (more on this later) and an R register to provide for automatic refresh of dynamic memory (again, more on this later.)
4. For our purposes, the IX and IY will be of considerable use; however, we will defer use of the others until an appropriate time later in the course.

#### IV. Overview of the Z80 instruction set

- A. We have noted that the Z80 architecture is derived from that of the 8080. This causes some peculiar patterns in the instruction set:

1. The 8080 uses instructions with one byte (8-bit) opcodes, possibly followed by 1-2 bytes of immediate data or direct address - yielding an instruction length of 1-3 bytes:

op\_\_code

or op\_\_code imm\_data

or op\_\_code imm\_data imm\_data

or op\_\_code address\_ address\_

2. Theoretically, the 8080 could have 256 different operations. However, it only has 244, leaving 12 of the possible opcodes unused.
3. The Z80 uses 8 of the unused 8080 opcodes for instructions not available on the 8080 - making a total of 252 instructions with a one-byte opcode. The remaining 4 codes serve as "windows" to an extended instruction set with opcodes of 2 or 3 bytes plus possibly 1-2 bytes of immediate data or address, producing instructions up to 4 bytes long:

\_window\_ op\_\_code

\_window\_ op\_\_code imm\_data

\_window\_ op\_\_code imm\_data imm\_data

\_window\_ op\_\_code address\_ address\_

\_window\_ op\_\_code op\_\_code

\_window\_ op\_\_code op\_\_code imm\_data

4. Our approach will be to begin our discussion with the one-byte opcodes. Then we move on to the 2-3 byte extensions.

5. SHOW TRANSPARENCY OF Z80 1-BYTE INSTRUCTIONS; HAND OUT

B. NOP 0000 0000; HALT 0111 0110

- C. The 8-bit LD group: Similar to VAX MOVB instruction. (Note: the VAX uses one instruction with a variety of modes/registers; Z80 uses a similar approach but lists each as a separate instruction.) Also, note that the flags (condition codes) are unaffected (contrast with the VAX)

1. Register addressing mode: 0100 0000 .. 0111 1111

a. Note that, in contrast to the VAX, the first register named is the destination and the second is the source. This is true both in assembly language and in machine language.

b. Note pattern to format: 01dddsss

000 = B; 001 = C; 010 = D; 011 = E; 100 = H; 101 = L;  
110 = (HL) (8080 mnemonic = M); 111 = A.

c. Note use of register indirect (HL) to address memory. To move data to/from a memory location using these instructions, one must first place the 16-bit address in the HL pair (using other instructions to be covered shortly.)

d. Slot that would be LD (HL) (HL) is HALT instead.

2. 8-bit immediate loads (two-byte instruction) 00\_\_110 \_value\_\_

3. Further addressing options with A only:

a. Register (other than HL) indirect: 000\_\_010

b. Absolute address: 0011\_010 \_addrLSB \_addrMSB (3 byte instruction)  
[Note byte reversed format: least significant byte of address is first, then most significant. This is true across the board for 16-bit values.]

D. 8-bit arithmetic/logical group: Note that destination operand is always A. This may be implicit or explicit in the assembly language syntax. (Explicit mention of A is needed in ADD, ADC, SBC because there are versions of these for 16-bit register pairs as we shall see later.) These instructions do set the flags (condition codes).

1. Register-mode addressing: 1000 0000 .. 1011 1111

Encoding pattern: 10ffffsss (fff = function)

- a. 000 ADD - 2's complement; can also be used for pair of BCD digits in conjunction with DAA (see discussion below).
- b. 001 ADC - adds a number plus carry from previous operation. (Like ADWC on VAX.)
- c. 010 SUB - again can be 2's complement or BCD.
- d. 011 SBC - like ADC
- e. 100 AND, 101 OR, 110 XOR - bitwise operation
- f. 111 CP = compare; A-source is used to set flags but isn't stored.

2. Immediate mode addressing: 11ffff110 \_value\_\_

E. 8-bit increment (00\_\_100); decrement (00\_\_101)

1. Note encoding: middle three bits specify register as with LD.

2. As on the VAX, these do set the flags.

F. 8-bit miscellaneous: 00\_\_111

1. RLCA - C <- 7 <--- 0

/<-----/

2. RRCA - 7 ---> 0 -> C

3. RLA - C <- 7 <--- 0  
/----->/

4. RRA - 7 ---> 0 -> C  
/<-----/

5. DAA (A register): This can be done immediately after an ADD or SUB involving the A register, and corrects the A register value and carry to what it should be if the values added were interpreted as a pair of BCD digits instead of an 8-bit binary number.

a. Consider what would happen if a pair of BCD digits were added as if binary - e.g.

11 + 22 = 33 - ok

14 + 66 = 7A - wrong - should be 80 (the first nibble should have carried into the second instead of becoming A)

18 + 68 = 80 - wrong - should be 86 (the internibble carry occurred, but the low nibble is 6 too low)

etc.

b. The Z80 flags include a half-carry flag H, which is set by ADD/SUB to record the inter-nibble carry. The DAA instruction uses this flag, together with the contents of the A register, to make a correction so that the ADD/SUB comes out correctly for BCD. For example, in the above cases:

11 + 22 = 33 - H is not set, both nibbles of A are valid BCD digits, so DAA would do nothing

14 + 66 = 7A - H is not set, but low nibble of A is not a valid BCD digit. DAA automatically adds 06 (to compensate for the 6 unused codes), yielding the correct result 80.

18 + 68 = 80 - H is set, so even though both nibbles are valid BCD digits DAA automatically adds 06, yielding the correct result 86.

(Note: DAA uses the C flag in conjunction with the upper nibble in a similar way.)

c. Thus, to add two 2-digit BCD numbers (say one in A, one in B), code:

```
ADD  A,B
DAA
```

To add two 4-digit BCD numbers (say one in B,C, other in D,E, with result in B,C), code:

```
LD   A,C
ADD  A,E
DAA
LD   C,A
LD   A,B
ADC  A,D
DAA
LD   B,A
```

d. DAA works with subtraction as well as addition. The N flag in the CPU records whether the last operation was add or subtract, and the DAA uses this to adjust accordingly.

6. CPL (1's complement A)



7. SCF, CCF: Set, clear carry. This can be used before operations like RLA, RRA, ADC or SBC to control (initial) value of the C flag.

#### G. 16-bit operations

1. Immediate loads to register pairs 00rr0001 \_LSBdata\_ \_MSBdata\_  
[Note byte-reversed format again]
2. HL load/store with direct memory address: 0010\_010 \_LSBaddr\_ \_MSBaddr\_
3. Add register pair to HL: 00rr1001
4. Increment/decrement register pair: 00rrf011
5. Load SP from HL 11111001

#### H. Conditional/unconditional transfers of control

1. As on the VAX, the Z80 sets certain condition code bits as a result of various operations, and these can then be tested for conditional jumps and other operations. The condition codes are called flags, and the register containing them is the F register, laid out as follows:

S Z x H x P/V N C

Meaning of the flags:

- a. S = sign bit of result. (Referred to in mnemonics below as P -> result was  $\geq 0$ , therefore S=0; M -> result was  $< 0$  therefore S=1). (Similar to VAX N bit. Note that Z80 has an N bit but it has a very different meaning.)
- b. Z = result was 0. (Same as VAX Z bit)
- c. P/V = parity or overflow. This flag is used for two different functions depending on the operation just performed:
  - i. After a bitwise logical instruction such as AND, OR, XOR, this flag is set if the result had even parity, and cleared if it had odd. (The VAX has no equivalent condition code bit.)
  - ii. After an arithmetic operation such as ADD the flag will be set to denote overflow. (Same as VAX V bit.)(Mnemonics use PE = parity even or PO = parity odd. There is no mnemonic for overflow; note that JP PE,\_\_\_ is equivalent to jump if overflow occurred.)
- d. C = carry out of arithmetic operation such as ADD or bit lost on a shift. (Similar to VAX C bit.)
- e. H and N are set on ADD/SUB type operations to retain half carry for use with DAA, and to record whether the operation was an ADD (N=0) or SUB (N=1).

2. Conditional jumps: 11ccc010 + two-byte direct address  
001cc000 + one-byte relative offset (Z80 unique)

Note assembler syntax for conditional form: JP cc, dest or JR cc, dest - where cc is one of the mnemonics:

NZ -> Z flag clear (result  $\neq 0$ )  
Z -> Z flag set (result = 0)  
NC -> C flag clear

```

C -> C flag set
PO -> parity odd: P/V flag clear
PE -> parity even: P/V flag set
P  -> S flag clear: result >= 0
M  -> S flag set: result <0

```

Note also that JR allows only some of these possibilities. This is because JP was an 8080 instruction, while JR is unique to the Z80. There were not enough one-byte opcodes left to give JR the same flexibility as JP.

3. Unconditional jumps: 11000011 + two-byte direct address  
                           11101001 : jump to address in HL  
                           00011000 + one-byte relative offset (Z80 unique)

4. CALLs (return address pushed on stack): 11cccc100 or 11001101 + 2-byte direct address).

Note that these can be unconditional or conditional as with jumps.  
 (A conditional JSR)

5. Subroutine returns (pop stack): 11cccc000 or 11001001. These can be unconditional or conditional as with jumps.

6. Resets (software interrupts) 11\_\_\_\_111. We will discuss these later.

7. DJNZ + one byte relative address: unique to Z80  
 (Decrements B - jumps if B<>0): 00010000

#### I. Stack operations, exchanges:

1. 16-bit pushes: 11rr0001 - where rr designates one of the pairs  
                           AF, BC, DE, HL
2. 16-bit pops: 11rr0101 - rr as above
3. 16-bit exchanges: 00001000 (unique to Z80) - exchanges two versions of AF  
                           11011001 (unique to Z80) - exchanges two versions of remaining registers  
                           11100011 - exchanges HL register with top item on stack (memory cell pointed to by SP)  
                           11101011 - exchanges DE register pair with HL pair

#### J. Input output:

1. OUT: 11010011 + port
2. IN:  11011011 + port
3. Interrupt enable disable: 1111\_011

#### K. Two byte opcodes with prefix = CB (all unique to Z80) (TRANSPARENCY)

1. Extended shift/rotate operations: CB + 00\_\_\_\_\_. On the 8080, only the A register could be shifted/rotated. This group allows shifts/rotates to be done on other registers or a memory cell pointed to by HL, plus adds three types of shift not found on 8080.
  - a. RLC, RRC: circular with 8-bits of register, CF also set (cf RLCA, RRCA above). Note that this group includes an operation on A, but for this register use the one-byte instructions RLCA, RRCA.
  - b. RL, RR rotate treating A + CF as 9 bits - see RLA, RRA above

- c. SLA, SRA: arithmetic shifts - propagate sign on right shift, shift in zero on left shift; CF always captures bit lost. (If needed on A register, must use this form since there is no one-byte form.)

SLA:           C <- 7 <--- 0 <- 0

              /---/  
SRA:           /-> 7 ---> 0 -> C

- d. SRL: logical right shift: shift in zero to sign; CF captures bit lost.

              0 -> 7 ---> 0 -> C

- 2. Bit test, reset, set operations: CB + 01\_\_\_\_, 10\_\_\_\_, 11\_\_\_\_

- a. Note format: oprrrbbb.
- b. On BIT, the bit of register rrr specified by bbb is tested, and the Z bit is set if tested bit was 0 (not set).
- c. SET sets bit bbb of register rrr to 1, and RES resets it to 0.
- d. On all of these, bit 7 is the MSB (sign), bit 0 the LSB.

- L. Two byte opcodes with first byte = DD, FD: allow use of IX, IY

- 1. DD + op: TRANSPARENCY - compare to basic one-byte instructions and note that IX occurs where HL occurs there, <IX+disp> where (HL) occurs.
  - a. The <IX+disp> form allows for a form of indexed-mode addressing (cf VAX modes A,C,E). The effective address for the operation is formed by fetching a byte from the instruction stream and adding it to IX to form an address. This is very useful for stepping through tables, etc.
  - b. Note that IX is only accessible through these multi-byte opcodes. Recall that IX (and IY) were not part of the original 8080 architecture.
- 2. DD + CB + op: TRANSPARENCY-compare to two-byte beginning with CB; again IX fills HL slots.
- 3. FD + op, FD + CB + op are similar to the above, save they use IY.

- M. Two byte opcodes with first byte = ED (TRANSPARENCY)

- 1. Extended IN, OUT with port number in C; data going to/from any register  
ED + 01\_\_000, 01\_\_001.  
  
(8080 IN, OUT instructions - also present on Z80 - transfer data to/from A and require port number as second byte of instruction)
- 2. 16-bit ADC, SBC allows 32-bit operations: ED + 01\_\_010: C bit is set by carry/borrow out of one 16-bit add/subtract and then can be brought into another.
- 3. Direct addressing 16-bit loads, stores for register pairs:  
ED + 01\_\_011
- 4. NEGate register A (2's comp): ED + 01000100

5. Control for more sophisticated Z80 interrupt mechanism (more on this later): ED + 0100\_101, 01\_\_110, 01\_\_0111.

6. "Nibble" oriented rotates: ED + 0110\_111. These rotate a nibble between the A register and the memory location pointed to by HL.

a. RLD: A Memory cell pointed to by HL

high	<--	low	<---	high	<---	low	<--
nibble		nibble		nibble		nibble	
-----							
(unchanged)							

b. RRD: A Memory cell pointed to by HL

high	-->	low	--->	high	--->	low	--
nibble		nibble		nibble		nibble	
-----							
(unchanged)							

7. Block transfer instructions: ED + 10\_\_\_\_\_:

- Common to all is the use of the BC pair as a byte counter, and HL and possibly DE as string pointers.
- The "non-R" forms perform a single byte transfer, adjusting the pointers (HL and possibly DE) up ("I" form) or down ("D"), then decrement BC.
- The "R" form repeats the operation of transfer until BC becomes 0 - i.e. it loops on the non-R form.
- LDxx allows movement of a string of bytes from a location pointed to by HL to a location pointed to by DE.
- CPxx allows comparison of a string of bytes at a location pointed to by HL to A. Comparison will terminate on the first match to the value in A, or when BC becomes 0 ("R" form).
- INxx and OTxx allow transfer of a string of bytes pointed to by HL to/from an IO port specified by C. For these operations, B alone is used as the byte counter.

## V. Calculating Z80 instruction timing and program run times

- In our previous work in programming, we have sometimes dealt with the matter of the execution times of programs from the standpoint of efficiency - seeking to write a program that runs in the shortest possible time.
- In the case of microprocessors, however, we frequently encounter situations in which our concern is with accuracy: knowing exactly how much time a given program will take to run. This is because the interface between the microprocessor and the outside world sometimes depends on precise timing relationships for the sending and receiving of information.
- Lab #9 is intended to get you into the matter of timing by having you program the MPF-I to behave as a stopwatch. Therefore, you will have to determine the exact amount of time needed for the execution of each section of the program you write.

D. On most microprocessor systems, timing is controlled by an external clock which is an oscillator that generates pulses at a fixed rate. Since all instruction times are a multiple of the clock period, this clock is normally crystal-controlled to ensure accuracy. On the MPF-I, the clock frequency is 1.79 mhz. This is about 30% less than the maximum permissible frequency for the chip used - 2.5 mhz. (Note: other versions of the Z80 can use clock frequencies up to 4.5 mhz.) Using a less than maximal clock frequency helps to ensure reliable system performance.

E. Given a knowledge of the clock frequency, one need only figure out the number of clock pulses needed for the execution of a given instruction to determine its execution time. This information is obtained from tables supplied by the microprocessor manufacturer.

1. For example, the instruction LD A,B requires four clock pulses to execute, and therefore takes  $4/1.79 \times 10^{-6} \text{ sec}^{-1} = 2.234637 \text{ microseconds}$ . (On a Z80 with a 4.5 mhz clock it would take  $4/4.5 \times 10^{-6} \text{ sec}^{-1} = .8888889 \text{ microseconds}$ .)

2. We will see how to use the necessary tables shortly.

F. What are the factors which give rise to the particular execution timing of a particular instruction? To answer this, we must examine how the Z80 actually executes an instruction, as well as the protocol for the Z80's communication with the outside world:

1. The execution time for a given instruction is composed of a number of major cycles, called M-cycles in the Z80 documentation. Normally, each M cycle involves the transfer of one byte of data between the Z80 and either memory or an IO port. Sometimes, however, an M cycle is dedicated to internal processing within the chip itself.

- a. For example, consider the instruction

ADD A,B - add the contents of register B to register A

This instruction is 1 byte long (just the opcode) and involves no further access to memory since all operands are in registers. Executing it, therefore, involves one major (M) cycle - that needed to fetch the instruction in the first place.

- b. Again, consider the instruction:

LD A, (nn) - load the A register with the byte contained at memory address nn.

This instruction occupies 3 bytes: 1 for the opcode and 2 for the address nn. Fetching this instruction therefore involves reading 3 bytes from memory. In addition, having fetched the instruction it becomes necessary to go to the designated memory address to fetch a value to load into A. This requires a 4th memory access. Thus, this instruction requires 4 major (M) cycles:

M1 - fetch the opcode byte  
M2 - fetch the low-order byte of the address word  
M3 - fetch the high-order byte of the address word  
M4 - fetch the operand

2. Each M cycle, in turn, requires anywhere from 3 to 6 clock pulses or T states for completion. For example, in the above M1 requires 4 T states and the remaining cycles require 3 each, for a total of 13 T states for the entire instruction.

3. To understand the relationship between M cycles and T states, we need to consider what the Z80 has to do to transfer a byte of data to/from memory or an IO port.

- a. The clock for the Z80 is a square wave. Each T state corresponds to a complete cycle of this square wave. Thus, there are two clock transitions per T-state.
  - i. The rising transition signals the end of one T-state and the start of the next.
  - ii. The falling transition occurs in the middle of the T-state.
- b. During the course of a data transfer, the Z80 generates a sequence of control signals that serve to synchronize the processor and the memory or external device. Each change in a control signal coincides with one of the transitions of the clock.
- c. Example: Memory read:

TRANSPARENCY - MEMORY READ/WRITE SEQUENCES (From Carr p29)

- i. To initiate a memory read, the Z80 places a 16-bit address on the address bus and asserts the control signals  $\overline{\text{MREQ}}$  and  $\overline{\text{RD}}$  (memory request and read.) Note that these signals are "low true": their asserted state is 0V, not 5V.
  - The address is put on the bus at the start of the first T-state of the M-cycle, on the rising transition of the clock.
  - The control lines are asserted in the middle of the T-state, on the falling clock transition. This allows time for the address to "settle" on the bus so that no device will misinterpret it when the control signals are asserted.
- ii. The two control signals remain asserted for a total of two clock times - until the middle of the third T-state. This allows time for the memory to recognize the command and respond by putting the data on the bus. The Z80 actually reads the data during the first half of the third T-state, just before deasserting the control signals.
- iii. Note that the address actually remains on the bus until the end of the third T-state. This is done so that the address bus does not change value during the time that the control signals are still asserted, lest the wrong memory see an address in transition and the control signals asserted and respond when it shouldn't.
- iv. Actually, this diagram assumes that the memory is able to get its data up on the bus not more than 1-1/2 clock periods after the request. Some memories may not be fast enough to do this. To allow such memories to function, the Z80 has a control line called  $\overline{\text{WAIT}}$  (low true). A slow memory may assert this to force the Z80 to inject one or more wait states ( $T_w$ ) before capturing the data on the third and final T-state. The timing tables we will use assume no wait states will occur (which is true for the MPF-I.)

d. Example: memory write

e. Example: I/O read write

TRANSPARENCY - I/O READ/WRITE SEQUENCES (from Carr page 30)

- f. Example: Opcode fetch - see Carr p. 27. The opcode fetch (M1) cycle is a special variant of the memory read which takes an additional T-state to allow for memory refresh. We will come back to this later.

G. To find out the number of clock pulses needed to execute a given instruction, one can consult an appropriate table.

1. Two places to look are

MPF-I user's Guide appendix C, beginning with page C14.  
Z80 Microprocessor Programming and Interfacing, appendix A.

2. In either case, the place to look is the column headed "Number of T states".

EXAMPLE: TRANSPARENCY SHOWING M CYCLES AND T STATES FOR A NUMBER OF INSTRUCTIONS FROM THE 8-BIT LOAD GROUP

3. Example: timing calculation for the following program segment:

LD	C,A	1 M cycle, 4 T-states
LD	B,0	2 M cycles, 7 T-states
LD	A,(BC)	2 M cycles, 7 T-states

4. Three cautions are in order, however:

a. Be sure you look at the right instruction. In the first example, LD A,B was an 8-bit load in which both source and destination were registers. The number of T states, and hence the timing, is radically different for other variants of LD - for example:

LD A,(HL) takes 7 T states - almost twice as long - since it needs to make an additional reference to memory to get one operand.  
LD SP,HL takes 6 T states, since it manipulates 16 bits.  
LD SP,IX takes 10 T states, since it is a 16 bit operation whose opcode is also two bytes long.

b. For some conditional jump instructions, the number of T states depends on whether or not the jump is taken. The same also holds for high-power looping instructions. For example, for JR NZ,(nn):

- i. If the jump is NOT taken, only 7 T states are needed.
- ii. If it is taken, another major cycle and 5 more T states (12 total) are needed to compute the relative jump address internally - since this requires a 16-bit add to the current PC value.

In reading the table, you will find two consecutive entries for such instructions - one of the branch is taken and one if it is not.

c. Understand that the times given in the table are best case times. Under certain circumstances, an instruction can take longer. Two things can prolong an instruction:

- i. If the memory chips in use are not able to read or write data quickly enough, additional wait T states will be needed whenever a reference to memory is made. The memory chip causes this by asserting the control line WAIT during the T state after that producing MREQ. The Z80 inserts TW wait states until the memory releases WAIT. (TRANSPARENCY) (This is not a problem on the MPF-I)
- ii. IO operations can be delayed if the IO port selected cannot respond immediately. Notice that in the IO timing one TW wait state is always inserted; the port can request more by asserting WAIT. (TRANSPARENCY)

In either case, however, the instruction time will be some even multiple of the clock period - the delay is effected by adding wait

states to the execution cycle.

- H. When writing programs in which exact timing is crucial you must balance both halves of every alternate path in your program. For example, suppose the following decision structure occurred in the middle of a segment whose exact time is critical:

```
IF A = B THEN
  C := 0
```

1. If timing were not a concern, one could code this as:

```
      CP B
      JR NZ, (SKIPCLR)
      LD C,0
SKIPCLR
```

The time for this segment would be:

```
A = B:      CP = 4 + JR (not taken) = 7 + LD = 7 - total 18
A <>B:      CP = 4 + JR (taken) = 12 = total 16
```

2. To balance both sides, one might code as follows:

```
      CP B
      JR NZ, (SKIPCLR)
      LD C,0
      JR (ENDIF)
SKIPCLR JR Z, (ENDIF) ; Cannot possibly be true since we came here on NZ
      JR Z, (ENDIF)

ENDIF
```

The time would now be:

```
A = B:      CP = 4 + JR (not taken) = 7 + LD = 7 + JR = 12-total 30
A <> B:      CP = 4 + JR (taken) = 12 + 2 JR's that can't be taken =
              2*7 - total 30
```

3. One way to handle such problems is to flowchart the program, computing the total number of T-states on each branch of the flowchart and making sure they are equal.
4. Obviously, one only does this if knowing the exact time of a segment is more important than keeping its size and time down! (As, for example, in simulating a clock.)

## VI. Z80 Pinouts

### A. Introduction

1. Thus far, we have looked at the Z80 primarily from a software standpoint: how one goes about programming it.
2. Now, we begin looking at the Z80 from a hardware standpoint: how one interfaces other devices to it.
3. We will do two things to accomplish this goal:
  - a. First, we will look at the pinouts and control signals of the Z80.
  - b. Second, we will look in detail at the MPF-I circuit diagrams to see how the MPF-I designers interfaced various devices to the bus.

- B. As we have noted previously, the Z80 has 40 pins: an 8-bit data bus, a 16-bit data bus, 14 control signals, and +5V and ground power connections.



## TRANSPARENCY: Z80 PINOUTS

- C. The 8-bit data bus consists of lines D0..D7, where D0 is the least significant bit. This bus is unique on the Z80 in that it is bidirectional (note the double-headed arrows.) The Z80 can send a byte of data to an external device over the bus, and an external device can send a byte of data to the Z80 over the bus.
1. This bus is used both for data transfers to/from memory and for IO operations.
  2. The Z80 connects to these pins via tri-state gates. When the Z80 is sending data over the bus, it drives each bit to either 5V or ground as appropriate. When it is receiving data over the bus, it presents a high-impedance load so that the sending device can drive it to the appropriate level. The Z80 will also "float" the bus (present a high impedance load to it) when it is allowing a direct memory access device to use the bus to communicate with memory, as we shall discuss later.
  3. External devices that can write to the bus must also connect to it via tri-state gates so that they do not interfere with the Z80 or other devices. Most chips that are meant to connect directly to microprocessor data busses (e.g. memories) have these tri-state gates built in.
- D. The 16 bit address bus consists of lines A0..A15, where A0 is the least significant bit. This is a one-way bus - Z80 to outside world.
1. For memory operations, the Z80 places a 16-bit address on the bus.
  2. For IO operations, the Z80 places an 8-bit port number on bits A0..A7. The high order 8 bits are not used for IO.
  3. Though the bus is one-directional, the Z80 connects to it via tri-state gates. This is so the Z80 can "float" the bus to allow direct memory access devices to take it over and address memory themselves.
- E. The output lines  $\overline{\text{MREQ}}$ ,  $\overline{\text{RD}}$ ,  $\overline{\text{WR}}$ , and  $\overline{\text{WAIT}}$  are used together for memory operations.
1. Note that these signals are low true: the asserted (true) state is represented by 5V and the unasserted state by 0V. This is the meaning of the bar over the mnemonics for the signals.
  2. When no bus operation is taking place, then, these lines are at 5V, which looks to external devices like a logic 1, though the meaning is "false". Memories look for a 0V condition (that looks like logic 0) in order to respond.
  3. Note that  $\overline{\text{MREQ}}$ ,  $\overline{\text{RD}}$ , and  $\overline{\text{WR}}$  are outputs from the Z80 while  $\overline{\text{WAIT}}$  is an input.  $\overline{\text{MREQ}}$ ,  $\overline{\text{RD}}$ , and  $\overline{\text{WR}}$  are connected to the bus via tri-state gates; when the Z80 is allowing a DMA device to control the address and data lines for direct transfer to/from memory, it also floats these lines to allow the device to control them as well. (The DMA device will follow the same protocol in controlling these lines that the Z80 would.) This however, does not make them input lines - when they are "floated", the Z80 ignores them.
  4. To do a memory read, the Z80:
    - a. Places a 16 bit address on A0..A15.

- b. "Floats" D0..D7.
- c. After allowing time for the bus to settle, it asserts  $\overline{\text{MREQ}}$  and  $\overline{\text{RD}}$ .
- d. The appropriate memory chip must now look up the requested byte and place it on the data bus. The Z80 will capture the data about 2 T-states after it asserts the request lines, and will then deassert the request lines. If the memory cannot respond this quickly, it must assert  $\overline{\text{WAIT}}$  (pull it to 0V) to cause the Z80 to insert extra Tw wait states before capturing the data.
- e. Finally, the Z80 will take the address off the bus.

REVIEW TRANSPARENCY: MEMORY TIMING (READ)

5. To do a memory write, the Z80:

- a. Places a 16 bit address on A0..A15.
- b. Places a byte of data on D0..D7.
- c. After allowing time for the bus to settle, it asserts  $\overline{\text{MREQ}}$  and  $\overline{\text{WR}}$ .
- d. The appropriate memory chip must now read the data byte from the bus and store it. If it cannot do so in the allowed time, it must assert  $\overline{\text{WAIT}}$  to request one or more Tw wait states.
- e. After about 2 T-states (plus any requested wait states), the Z80 will deassert the control signals, while still leaving address and data valid. (The memory may use this transition to actually trigger the data transfer.)
- f. Finally, the Z80 will take the address and data off the bus.

REVIEW TRANSPARENCY: MEMORY TIMING (WRITE)

F. The control signal  $\overline{\text{IORQ}}$  is used in conjunction with  $\overline{\text{RD}}$ ,  $\overline{\text{WR}}$  and  $\overline{\text{WAIT}}$  for IO transfers.

- 1. IO reads and writes follow much the same protocol as memory reads and writes, except that the control line  $\overline{\text{IORQ}}$  is used in place of  $\overline{\text{MREQ}}$ . This distinction is what causes IO devices to respond, rather than memories.
- 2. Another distinction between IO operations and memory operations is the use of an 8 bit port number on A0..A7 in place of a 16 bit address using the entire address bus.
- 3. IO devices may use  $\overline{\text{WAIT}}$  just as memories can to request additional time. A distinction between IO and memory operations is that all IO operations include at least one Tw wait state automatically.

REVIEW TRANSPARENCY: IO TIMING

G. The  $\overline{\text{M1}}$  signal is an active low output that is asserted (0 Volts) when the Z80 is in the opcode fetch (M1) major cycle of an instruction, and is unasserted (5 Volts) at all other times. In the case of multi-byte opcodes (window byte + additional byte),  $\overline{\text{M1}}$  is low for each byte fetched.

TRANSPARENCY: M1 CYCLE (INSTRUCTION FETCH)

- H. The  $\overline{\text{RFSH}}$  output is used as part of a very nice feature of the Z80: automatic refresh for dynamic memory. We will discuss this later when we talk about memory systems. (Note: this is not of interest on the MPF-I, since it uses static RAM, not dynamic RAM.)

#### NOTE ON M1 CYCLE TRANSPARENCY

- I. The  $\overline{\text{INT}}$  and  $\overline{\text{NMI}}$  lines are active low inputs to the Z80 that allow an external device to initiate an interrupt. We will discuss the details of interrupts later in the course. For now, we note the following:
1. The  $\overline{\text{INT}}$  line provides a very flexible interrupt mechanism whereby the interrupting device can cause the Z80 to execute a specific interrupt service routine for that device. This kind of interrupt can be enabled and disabled under software control by using the EI and DI instructions. In particular, interrupt service routines normally run with interrupts disabled so that the servicing of one interrupt will not be interrupt by another request.
  2. The  $\overline{\text{NMI}}$  line provides a high priority interrupt mechanism that always executes an interrupt service routine at a specific address in memory: 0066H. Unlike  $\overline{\text{INT}}$ , interrupts requested via  $\overline{\text{NMI}}$  cannot be disabled - a non-maskable interrupt request will always be honored at the end of the currently-executing instruction. Moreover, if requests are pending on both  $\overline{\text{INT}}$  and  $\overline{\text{NMI}}$ , the  $\overline{\text{NMI}}$  request will have priority. This provides a way of guaranteeing fast response to certain kinds of external event, such as impending power failure.
  3. In the case of  $\overline{\text{INT}}$  (but not  $\overline{\text{NMI}}$ ) the Z80 will acknowledge the interrupt request when it is able to do so, allowing the external device to provide further information as to the action to take.

The Z80 acknowledges an interrupt by asserting both  $\overline{\text{IORQ}}$  and  $\overline{\text{M1}}$  at the same time. This is a cue for the interrupting device to place additional data on the bus.

#### TRANSPARENCY: INT TIMING

For now, the importance of this is that  $\overline{\text{IORQ}}$  does not always mean that an IO transfer is being done, and IO devices cannot therefore look

only at the address bus and  $\overline{\text{IORQ}}$  to see if they are being addressed. In particular:

$\overline{\text{IORQ}}$  and  $\overline{\text{M1}}$  indicates that an interrupt is being acknowledged

$\overline{\text{IORQ}}$  and  $\overline{\text{RD}}$  indicates that a read from the port addressed by A0..A7 is being done

$\overline{\text{IORQ}}$  and  $\overline{\text{WR}}$  indicates that a write to the port addressed by A0..A7 is being done

(We will see the effects of this when we look at the MPF-I's circuit diagrams.)

- J.  $\overline{\text{BUSERQ}}$  and  $\overline{\text{BUSAK}}$  are input lines to the Z80 that provide a way for direct-memory-access devices to gain control of the system bus.
1. Certain kinds of IO devices have data transfer rates that approaches the maximum speed of the memory system. For example, a floppy disk may have a transfer rate of 250K bytes/sec - or one byte every

4 microseconds. (Hard disks are even faster.) On a 2.5 mhz Z80, each M-cycle takes almost 2 microseconds; so the floppy disk would require a transfer every other M-cycle. This is probably too fast for data transfer under program control (though it might be possible using the INDR instruction.) The controllers for such devices typically transfer data directly from the device to memory (or vice versa) without CPU intervention.

2. The  $\overline{\text{BUSRQ}}$  line can be asserted by a DMA controller to indicate that it needs to use the system bus for a data transfer. At the end of the current M-cycle, the Z80 will:
  - a. Float its data, address, and control lines so that the external device can control them.
  - b. Assert  $\overline{\text{BUSAk}}$ , to let the device know that it can use the bus.
  - c. When the device is through with the bus, it releases  $\overline{\text{BUSRQ}}$  to allow the Z80 to resume using the bus.

#### TRANSPARENCY: $\overline{\text{BUSRQ}}$ TIMING

3. We will come back to this later.

- K. The  $\overline{\text{HALT}}$  output from the Z80 is asserted when the processor is halted - i.e. it has executed the HALT instruction. When the CPU is halted, it stops fetching and executing instructions, but it will respond to

interrupts ( $\overline{\text{NMI}}$  always,  $\overline{\text{INT}}$  only if enabled.) On some systems,  $\overline{\text{HALT}}$

is wired to  $\overline{\text{NMI}}$ ; thus, if a halt instruction is executed either accidentally or deliberately an immediate non-maskable interrupt will occur, restarting the processor.

- L. The  $\overline{\text{RESET}}$  line provides a mechanism for orderly system start up when power is first applied to a system.
  1. When the power is first turned on to any digital system, the various flip flops in the system will be in a random, unpredictable state. This will also be true of the data stored in semiconductor RAM.
  2. Many systems have a circuit that functions when power is first applied, asserting a system reset line for a few milliseconds or so. This reset line is then applied to all of the devices in the system, including the CPU, to initialize them.
  3. When the  $\overline{\text{RESET}}$  input to the Z80 is asserted, the Z80's program counter is set to 0, and it begins executing whatever program is loaded there. Thus, Z80-based systems typically reserve their lowest memory addresses for ROM, with a system initialization program beginning at location 0.
  4. On the MPF-I,  $\overline{\text{RESET}}$  is asserted under two conditions:
    - a. At power-up.
    - b. When the reset button on the keyboard is pressed.
- M. The clock input (O/) must be connected to a source of square waves of the appropriate frequency (e.g. 2.5 or 4.5 mhz).
- N. +5V and ground connections complete the Z80 pinouts.

## VII. MPF-I circuit diagrams

A. As a concrete example of how various devices can be interfaced to the Z80 bus, we consider the circuit diagrams for the MPF-I. These are five pages long in all, in appendix D of the MPF-I User's Manual.

1. Sheet 1 is the main circuit diagram
2. Sheets 2,3 detail display and tape recorder circuits
3. Sheet 4 details the keyboard circuit
4. Sheet 5 details the power supply circuits

B. Sheet 1 - the main circuit                      TRANSPARENCY

### 1. Clock:

- a. An oscillator at upper left-hand corner, consisting of two inverters, two resistors, and a crystal, produces a 3.58 mhz raw clock.
- b. This is applied to the clock input of two D flip flops (U11 - a 74LS74 dual D flip-flop). The lower of these is part of the reset circuit and will be considered later.
- c. The upper D flip flop has its Q' output wired to its D input. Thus, its state changes on every clock pulse. What is the effect of this? (ASK)
- d. It divides the clock frequency by 2 - the flip flop changes state 3.58E6 times per second, which means that it goes through a complete cycle of state changes (lo to hi to lo again) 1.59E6 times per second - yielding the desired 1.59 mhz square wave clock for the Z80.
- e. This signal is applied to the O/ input on the Z80, and is also made available on the system bus (though none of the other MPF-I devices use it.)

### 2. Reset:

- a. The other half of the U11 dual D flip-flop is used for system reset. It gets its clock from the same oscillator as described above. This means that its output be updated to reflect the current value of its input 3.58E6 times per second.
- b. The Q output of this flip flop is connected to the RESET input of the Z80. Thus, a 0 in the flip flop will cause a system reset. The normal operating condition of the system is for this flip flop to contain a 1.
- c. The D input to this flip flop is normally held high by a 10K resistor to +5V. There are two ways it can be pulled low (putting the flip flop in the zero state and resetting the system):
  - i. The reset switch on the keyboard can ground the D input.
  - ii. There is a 4.7 uf capacitor from the D input to ground. When the system is powered down, this capacitor is discharged. At power up, it takes a while for this capacitor to charge up through the 10K resistor. (The time constant is about 47 ms) Thus, the D input to the flip flop is held low during the first few milliseconds after the system is powered up, forcing a power-up reset of the Z80 as desired. As the capacitor charges up, its value eventually crosses the threshold recognized as the 0 to 1 transition by the flip flop, at which point the

flip flop goes to 1 and stays there for normal operation,.

- d. The Q' output of this flip flop is connected to the reset inputs of the 8255 IO ports shown on sheet 2 (which we will discuss later). Unlike the Z80, these devices require an active HIGH reset signal; thus, when Q is low and the Z80 is being reset, Q' will be high and the ports will also be reset, as desired.

3.  $\overline{\text{WAIT}}$ ,  $\overline{\text{BUSRQ}}$ , and  $\overline{\text{INT}}$ :

- a. These inputs are not used by the MPF-I. Since they are active low, they are tied to +5V by 10K pullup resistors to prevent spurious signalling.
- b. They are connected to the system bus connector so that external devices can use them. Although it is not shown in the diagram, the  $\overline{\text{BUSAK}}$  output is also put on the bus so that devices using  $\overline{\text{BUSRQ}}$  can know when the Z80 has responded to their request.

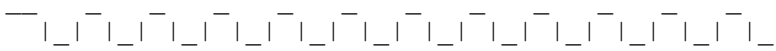

4.  $\overline{\text{NMI}}$ : This input is used by the MPF-I as part of a circuit to implement a single step/program breakpoint feature in the monitor.

- a. One nice debugging feature of the monitor is the S command. This will prompt the user for an address

/\_S\_\ =

will execute the instruction at that address, and will then return to the monitor prompt, allowing the user to examine registers etc. Repeated uses of this command make it possible to step through a program one instruction at a time.

- b. Implementing a feature like this requires hardware support. On many CPU's, this takes the form of a flag in the condition codes which the user can set to cause the CPU to trap after each instruction. However, the Z80 lacks this feature.
- c. To implement single stepping, the MPF-I uses one bit of one of its IO ports as a break signal (active low). This line is normally held high; the monitor outputs a 0 to this bit as part of the setup for stepping the program. (The logic for this is on sheet 2, which we will look at later.)
- d. This  $\overline{\text{break}}$  signal from the port is applied to the reset input of a 74LS90 divide-by-10 counter. When it is high (the normal state), the counter is held in the zero state. Because of the inverter between the counter output and the  $\overline{\text{NMI}}$ , this holds  $\overline{\text{NMI}}$  high (inactive.)
- e. When  $\overline{\text{break}}$  goes low, the counter starts running. The counter clock is connected to the  $\overline{\text{M1}}$  output of the Z80, so it counts M1 states - i.e. instructions executed (one M1 state for opcode fetch per instruction.)
- f. With the connections shown, the counter functions as a divide-by-5 counter - i.e. its input and output waveforms are:

input:      
output:   

Since the output of the counter is inverted before being applied to

$\overline{\text{NMI}}$ , this means that  $\overline{\text{NMI}}$  will go low - and an interrupt will be triggered, on the fifth M1 cycle after the monitor sets the bit in the port. The software is arranged, of course, so that this instruction will be the user instruction to single step (the other

4 being monitor code); since  $\overline{\text{NMI}}$  is asserted during its opcode fetch, the interrupt is actually taken after it finishes execution as desired. (Of course, the monitor must quickly reset the break bit to 1 to prevent the interrupt service routine from itself being interrupted!)

5.  $\overline{\text{HALT}}$ : This output is connected through a driver transistor to a red LED. When the CPU is running ( $\overline{\text{HALT}}$  deasserted or high), the LED will be lit; should a halt instruction be executed the LED will go out.

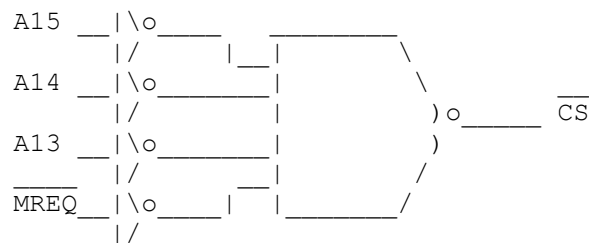
6. The memory system.

- a. The on-board memory consists of three chips, plus a slot for a fourth: 1-2 ROMs and 2 RAMs. (We actually only have on ROM on our MPF-I's, the other slot being a BASIC option we did not buy. However, we will discuss the hardware as if the ROM were there.)
- b. Each of the memory chips is organized into 8 bit bytes; therefore, each connects to all 8 bits of the data bus D0..D7.
- c. Each of the ROM's is an 8K chip; therefore, each must receive the low order 13 bits of the address bus: A0..A12. The RAM's are only 2K each, so they only get 11 bits: A0..A10. These low order address bus bits are decoded on the chip to access the correct byte.

- d. The high-order bits of the address bus are decoded externally to the memories, and are used to derive the  $\overline{\text{CS}}$  and  $\overline{\text{OE}}$  enable inputs to them. These active-low inputs are used by the memories to decide whether or not they should respond to activity on the bus; only if  $\overline{\text{CS}}$  is low, will the chip respond. The decoder circuitry also looks at the  $\overline{\text{MREQ}}$  control line, since the memory chips should only respond during a memory operation, regardless of the state of the address bus. The desired truth tables for the various chip selects is as follows:

Slot	Assigned addresses	$\overline{\text{MREQ}}$	A15..A11	$\overline{\text{CS}}$
U2	0000-1FFFFH	0	000xx	0
		0	all others	1
		1	xxxxx	1
U3	2000-3FFFFH	0	001xx	0
		0	all others	1
		1	xxxxx	1
U4	F000-F7FFFH	0	11110	0
		0	all others	1
		1	xxxxx	1
U5	F800-FFFFFH	0	11111	0
		0	all others	1
		1	xxxxx	1

These  $\overline{\text{CS}}$  values could be derived by using NAND gates - e.g. for U2:



- e. However, the MPF-I uses far fewer chips by using a 74LS138 1 out of 8 decoder chip to handle memory decoding.

TRANSPARENCY: 74LS138 diagram and truth table

- i. The 74LS138 has 6 inputs and 8 outputs.
  - ii. The outputs are active low. At most 1 of the 8 outputs can be low at any one time. Note that this is convenient for use with memory or IO chips whose chip select input is active low. (In fact, the memories are made this way BECAUSE decoders like the 74LS138 have active low outputs. The decoders, in turn, have active low outputs because they are built from NAND gates.)
  - iii. Three of the inputs are decoded to select one of the 8 outputs.
  - iv. The remaining three inputs are enables, which are anded together; i.e. all must have the proper value for the selected output line to go low.
    - Two of the enables are active low: unless they are both low, none of the outputs is low.
    - One of the enables is active high: unless it is high, none of the outputs is low.
- f. The memory decoder 74LS138 on the MPF-I (U6) is connected as follows:
- i. Its active high enable input is tied to +5V, and its two active low enables are both tied to  $\overline{\text{MREQ}}$ . Thus, its selected output is enabled whenever a memory operation is being called for.
  - ii. Its three decoded inputs are connected to address bus bits A15 .. A13. Thus, the first output line is selected when the high order three bits of the address are 000; this is routed to  $\overline{\text{CS}}$  on the ROM in U2. The second line goes to  $\overline{\text{CS}}$  on the ROM in U3; this is selected when the high order bits of the address are 001. The last output line is low when the high order bits of the address bus are 111, which means that RAM is being addressed; however, more decoding is needed to select a specific RAM chip.
- g. The 74LS138 output that is selected when RAM is addressed goes as an enable to another decoder that decodes two more bits of the address bus (A12..A11) to select the correct RAM chip. The network of jumpers at the right hand side allows for connections to be changed if different size RAMs are used or a RAM is put in the ROM slot U3.
- h. The RAM chips have an additional control input  $\overline{\text{WE}}$ , which the selected chip uses to decide whether to do a read or a write.  $\overline{\text{WE}}$  is a low true write enable; when it is low, the chip does a write, otherwise it does a read. Note that  $\overline{\text{WE}}$  is simply connected to  $\overline{\text{WR}}$



on the Z80 bus; if a read operation is in progress the chip need only know that  $\overline{WR}$  is not asserted - it need not actually see  $\overline{RD}$  asserted.

## 7. The IO system.

- a. The MPF-I has a total of 43 bits of IO to/from various devices. These are provided for by two 8255 chips, each of which has 3 eight-bit ports for a total of 48 bits. (Five are unused.) Each 8255 interfaces to the data bus and the two low-order address bus bits, the latter being used to select one of the 3 ports or the chip's 8-bit control register. (We will look at the 8255 later when we get to parallel IO.) Each 8255 also has a low true chip select, whose derivation we

consider now. (Note off sheet connections for  $\overline{CS1}$  and  $\overline{CS2}$ .)

- b. One of the 8255's is assigned port addresses 80-83, and the other 90-93. Since port numbers are eight bits, and the chip itself decodes two bits to select from among the ports on the chip, the external circuitry must decode 6 address bits for chip select ( $A7..A2$ ).

- c. Also, as with memory we only want to enable an IO chip when an IO operation is in progress. In this case, however, we must look at two control lines:  $\overline{IORQ}$  and  $\overline{M1}$ .

This is because  $\overline{IORQ}$  is not only asserted during an IO operation, but also during an interrupt acknowledge; however, in the latter case  $\overline{M1}$  is also asserted. Thus, for an IO operation to be selected it must be the case that  $\overline{IORQ}$  is low and  $\overline{M1}$  is HIGH.

- d. It would seem, then, that 8 lines must be decoded to derive chip select for the 8255's: 6 address lines ( $A7..A2$ ) and two control lines. However, the MPF-I uses a single 74LS138 (U7) that only decodes  $A7..A4$  plus the control lines. ( $A3$  and  $A2$  are ignored.) This means that the 8255's will actually respond to any addresses in the ranges 80..8F or 90..9F. (This is not a problem, since 256 ports is far more than the number needed. This is a common trick to save decoder logic.)

- i.  $\overline{M1}$  is connected to the active high enable, and  $\overline{IORQ}$  to one active low enable. This guarantees that the IO chips will only respond to IO operations, as required.
- ii. Address bus bit  $A7$  is inverted and connected to the other active low enable. This guarantees that the chips will only respond to port addresses  $\geq 80$ .
- iii. Finally, bits  $A6..A4$  are connected to the selection inputs, and the output lines corresponding to 000 and 001 are used to drive

$\overline{CS1}$  and  $\overline{CS2}$ , causing the 8255's to respond to IO operations on port addresses of the form 1000 xxxx and 1001 xxxx as required.

C. Sheet 2 - the IO ports                      TRANSPARENCY

D. Sheet 4 - the keyboard matrix            TRANSPARENCY