29F10

Microprogram Controller

Description

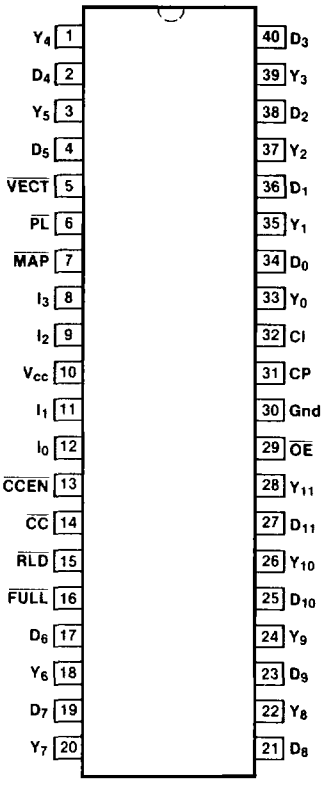
The 29F10 is a high-speed bipolar microprogram controller. It is intended for use in controlling the sequence of execution of microinstructions stored in microprogram memory. The 29F10 provides a 12-bit address during each clock cycle. The address comes from one of four sources: 1) Direct input from D_0-D_{11}; 2) A register/counter; 3) A microprogram counter; or 4) A five-deep LIFO stack. Address outputs are 3-state for maximum versatility.

The microprogram controller is compatible with FACT (Fairchild Advanced Schottky TTL) devices and can be used along with FAST parts in microprogrammed systems to minimize cycle times.

- **Addresses up to 4096 Words of Microcode**
- **Directly Loadable Down-Counter for Counting Loop Iterations**
- **Provides Count Capacity of 4096**
- **An Up Counter Providing Sequential MicroInstruction Execution**
- **A 5-Deep Push/Pop LIFO Stack Providing Subroutine Linkage and Branch Capabilities**
- **Registers are all Positive Edge-Triggered**
- **Plug-in Replacement for Standard 2910**

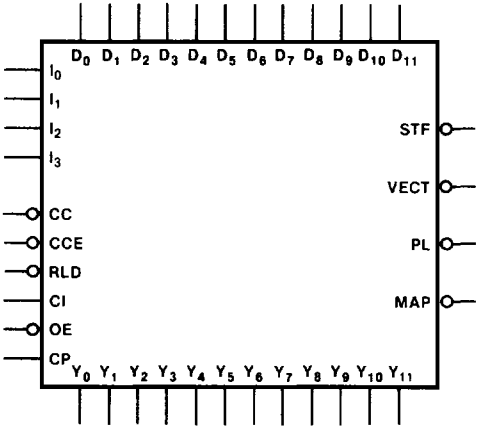
Ordering Code: See Section 5

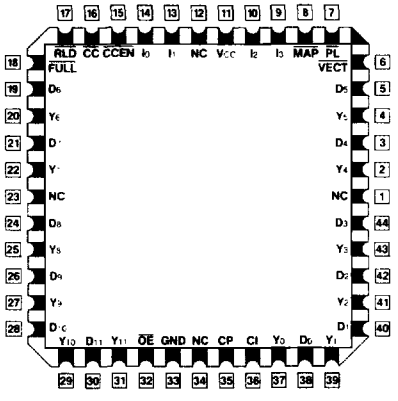
Connection Diagrams



Pin Assignment for DIP

Logic Symbol





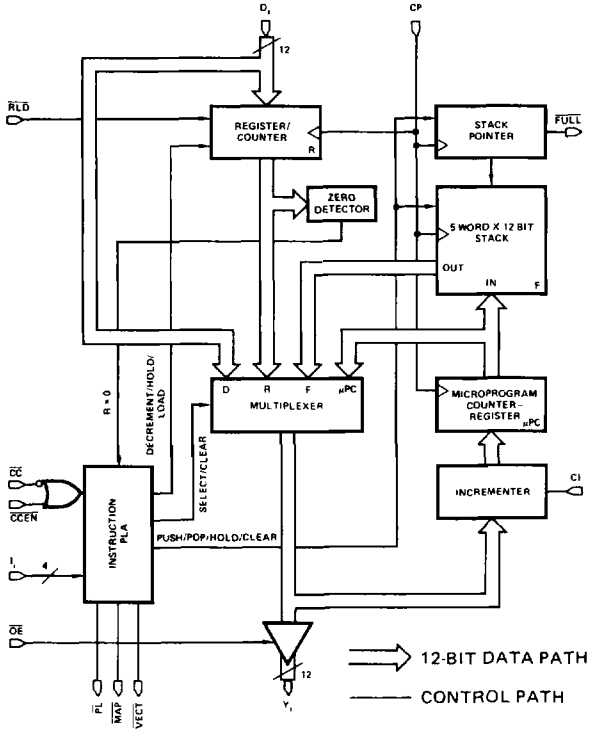
Pin Assignment for LCC and PCC

Input Loading/Fan-Out: See Section 3 for U.L. definitions

Pin Names	Description	29F(U.L.) HIGH/LOW
D_i	Direct Input	0.5/0.225
I_i	Instruct Bit	0.5/0.225
$\overline{CC}$	Condition Code	0.5/0.225
$\overline{CCEN}$	Condition Code Enable	0.5/0.225
$\overline{CI}$	Carry-In	0.5/0.225
$\overline{RLD}$	Register Load	0.5/0.225
$\overline{OE}$	Output Enable	0.5/0.450
CP	Clock Pulse	1.0/0.788
Y_0-Y_{11}	Microprogram Address Bits	0.5/0.225
$\overline{FULL}$	Status Full	0.5/0.225
$\overline{PL}$	Pipeline Address Enable	0.5/0.225
$\overline{MAP}$	Map Address Enable	0.5/0.225
$\overline{VECT}$	Vector Address Enable	0.5/0.225

4

Block Diagram



Instruction Set

The 29F10 provides sixteen instructions which select the address of the next microinstruction to be executed. Four of the instructions are unconditional—their effect depends only on the instruction. Ten of the instructions have an effect which is partially controlled by an external, data-dependent condition. Three of the instructions have an effect which is partially controlled by the contents of the internal register/counter. The instruction set is shown in Table 1. In this discussion it is assumed that CI is tied HIGH.

In the ten conditional instructions, the result of the data-dependent test is applied to $\overline{CC}$. If the $\overline{CC}$ input is LOW, the test is considered to have been passed and the action specified in the name occurs; otherwise, the test has failed and an alternate operation (often simply the execution of the next sequential microinstruction) occurs. Testing of $\overline{CC}$ may be disabled for a specific microinstruction by setting $\overline{CCEN}$ HIGH, which unconditionally forces the action specified in the name; that is, it forces a pass. Other ways of using $\overline{CCEN}$ include (1) tying it HIGH, which is useful if no microinstruction is data-dependent; (2) tying it LOW if data-dependent instructions are never forced unconditionally; or (3) tying it to the source of 29F10 instruction bit I_0, which leaves instructions 4, 6, and 10 as data-dependent but makes others unconditional. All of these tricks save one bit of microcode width.

The effect of three instructions depends on the contents of the register/counter. Unless the counter holds a value of zero, it is decremented; if it does hold zero, it is held and a different microprogram next address is selected. These instructions are useful for executing a microinstruction loop a known number of times. Instruction 15 is affected both by the external condition code and the internal register/counter.

Perhaps the best technique for understanding the 29F10 is to simply take each instruction and review its operation. In order to provide some feel for the actual execution of these instructions, Figure a is included and depicts examples of all sixteen instructions.

The examples given in Figure a should be interpreted in the following manner: The intent is to show microprogram flow as various microprogram memory words are executed. For example, the CONTINUE instruction, instruction number 14, as shown in Figure a, simply means that the contents of microprogram memory word 50 are executed, then the contents of word 51 are executed. This is followed by the contents of microprogram memory word 52 and the contents of microprogram memory word 53. The microprogram addresses used in the examples were arbitrarily chosen and have no meaning other than to show instruction flow. The exception to this is the first example, JUMP ZERO, which forces the microprogram location counter to address ZERO. Each dot refers to the time that the contents of the microprogram memory word are in the pipeline register. While no special symbology is used for the conditional instructions, the text to follow will explain what the conditional choices are in each example.

Instruction 0, JZ (JUMP and ZERO, or RESET) unconditionally specifies that the address of the next microinstruction is zero. Many designs use this feature for power-up sequences and provide the power-up firmware beginning at microprogram memory word location 0.

Instruction 1 is a CONDITIONAL JUMP-TO-SUBROUTING via the address provided in the pipeline register. As shown in Figure a, the machine might have executed words at address 50, 51, and 52. When the contents of address 52 are in the pipeline register, the next address control function is the CONDITIONAL JUMP-TO-SUBROUTINE. Here, if the test is passed, the next instruction executed will be the contents of microprogram memory location 90. If the test has failed, the JUMP-TO-SUBROUTINE will not be executed; the contents of microprogram memory location 53 will be executed instead. Thus, the CONDITIONAL JUMP-TO-SUBROUTINE instruction at location 52 will cause the instruction either in location 90 or in location 53 to be executed next. If the TEST input is such that location 90 is selected, value 53 will be pushed onto the internal stack. This provides the return linkage for the machine when the subroutine beginning at location 90 is completed. In this example, the subroutine was completed at location 93 and a RETURN-FROM-SUBROUTINE would be found at location 93.

Instruction 2 is the JUMP MAP instruction. This is an unconditional instruction which causes the $\overline{MAP}$ output to be enabled so that the next

microinstruction location is determined by the address supplied via the mapping PROMs. Normally, the JUMP MAP instruction is used at the end of the instruction fetch sequence for the machine. In the example of Figure a, microinstructions at locations 50, 51, 52, and 53 might have been the fetch sequence and at its completion at location 53, the jump map function would be contained the pipeline register. This example shows the mapping PROM outputs to be 90; therefore, an unconditional jump to microprogram memory address 90 is performed.

Instruction 3, CONDITIONAL JUMP PIPELINE, derives its branch address from the pipeline register branch address value. This instruction provides a technique for branching to various microprogram sequences depending upon the test condition inputs. Quite often, state machines are designed which simply execute tests on various inputs waiting for the condition to come true. When the true condition is reached, the machine then branches and executes a set of microinstructions to perform some function. This usually has the effect of resetting the input being tested until some point in the future. Figure a shows the conditional jump via the pipeline register address at location 52. When the contents of microprogram memory word 52 are in the pipeline register, the next address will be either location 53 or location 30 in this example. If the test is passed, the value currently in the pipeline register (30) will be selected. If the test fails, the next address selected will be contained in the microprogram counter which, in this example, is 53.

Instruction 4 is the PUSH/CONDITIONAL LOAD COUNTER instruction and is used primarily for setting up loops in microprogram firmware. In Figure a, when instruction 52 is in the pipeline register, a PUSH will be made onto the stack and the counter will be loaded based on the condition. When a PUSH occurs, the value pushed is always the next sequential instruction address. In this case, the address is 53. If the test fails, the counter is not loaded; if it is passed, the counter is loaded with the value contained in the pipeline register branch address field. Thus, a single microinstruction can be used to set up a loop to be executed a specific number of times. Instruction 8 will describe how to use the pushed value and the register/counter for looping.

Instruction 5 is a CONDITIONAL JUMP-TO-SUBROUTINE via the register/counter or the contents of the PIPELINE register. As shown in Figure a, a PUSH is always performed and one of two subroutines executed. In this example, either the subroutine beginning at address 80 or the subroutine beginning at address 90 will be performed. A return-from-subroutine (instruction number 10) returns the microprogram flow to address 55. In order for this microinstruction control sequence to operate correctly, both the next address fields of instruction 53 and the next address fields of instruction 54 would have to contain the proper value. Let's assume that the branch address fields of instruction 53 contain the value 90 so that it will be in the 29F10 register/counter when the contents of address 54 are in the pipeline register. This requires that the instruction at address 53 load the register/counter. Now, during the execution of instruction 5 (at address 54), if the test failed, the contents of the register (value = 90) will select the address of the next microinstruction. If the test input passes, the pipeline register contents (value = 80) will determine the address of the next microinstruction. Therefore, this instruction provides the ability to select one of two subroutines to be executed based on a test condition.

Instruction 6 is a CONDITIONAL JUMP VECTOR instruction which provides the capability to take the branch address from a third source heretofore not discussed. In order for this instruction to be useful, the 29F10 output, $\overline{\text{VECT}}$ is used to control a 3-state control input of a register, buffer, or PROM containing the next microprogram address. This instruction provides one technique for performing interrupt type branching at the microprogram level. Since this instruction is conditional, a pass causes the next address to be taken from the vector source, while failure causes the next address to be taken from the microprogram counter. In the example of Figure a, if the CONDITIONAL JUMP VECTOR instruction is contained at location 52, execution will continue at vector address 20 if the $\overline{\text{CC}}$ input is LOW and the microinstruction at address 53 will be executed if the $\overline{\text{CC}}$ input is HIGH.

Instruction 7 is a CONDITIONAL JUMP via the contents of the 29F10 REGISTER/COUNTER or the contents of the PIPELINE register. This instruction is very similar to instruction 5; the conditional

jump-to-subroutine via R or PL. The major difference between instruction 5 and instruction 7 is that no push onto the stack is perfomed with instruction 7. Figure a depicts this instruction as a branch to one of two locations depending on the test condition. The example assumes the pipeline register contains the value 70 when the contents of address 52 are being executed. As the contents of address 53 are clocked into the pipeline register, the value 70 is loaded into the register/counter in the 29F10. The value 80 is available when the contents of address 53 are in the pipeline register. Thus, control is transferred to either address 70 or address 80 depending on the test condition.

Instruction 8 is the REPEAT LOOP, COUNTER ZERO instruction. This microinstruction makes use of the decrementing capability of the register/counter. To be useful, some previous instruction, such as 4, must have loaded a count value into the register/counter. This instruction checks to see whether the register/counter contains a non-zero value. If so, the register/counter is decremented, and the address of the next microinstruction is taken from the top of the stack. If the register counter contains zero, the loop exit condition is occuring; control falls through to the next sequential microinstruction by selecting μPC; the stack is POPed by decrementing the stack pointer, but the contents of the top of the stack are thrown away.

An example of the REPEAT LOOP, COUNTER ZERO instruction is shown in Figure a. In this example, location 50 most likely would contain a PUSH/CONDITIONAL LOAD COUNTER instruction which would have caused address 51 to be PUSHed on the stack and the counter to be loaded with the proper value for looping the desired number of times.

In this example, since the loop test is made at the end of the instructions to be repeated (microaddress 54), the proper value to be loaded by the instructions at address 50 is one less than the desired number of passes through the loop. This method allows a loop to be executed 1 to 4096 times. If it is desired to execute the loop from 0 to 4095 times, the firmware should be written to make the loop exit test immediately after loop entry.

Single-microinstruction loops provide a highly efficient capability for executing a specific microinstruction a fixed number of times. Examples include fixed rotates, byte swap, fixed point multiply, and fixed point divide.

Instruction 9 is the REPEAT PIPELINE REGISTER, COUNTER ZERO instruction. This instruction is similar to instruction 8 except that the branch address now comes from the pipeline register rather than the file. In some cases, this instruction may be thought of as a one-word file extension; that is, by using this instruction, a loop with the counter can still be performed when subroutines are nested five deep. This instruction's operation is very similar to that of instruction 8. The differences are that on this instruction a failed test condition causes the source of the next microinstruction address to be the D inputs, and when the test condition is passed, this instruction does not perform a POP because the stack is not being used.

In the example of Figure a, the REPEAT PIPELINE, COUNTER ZERO instruction is instruction 52 and is shown as a single microinstruction loop. The address in the pipeline register would be 52. Instruction 51 in this example could be the LOAD COUNTER AND CONTINUE instruction (number 12). While the example shows a single microinstruction loop, by simply changing the address in a pipeline register, multi-instruction loops can be performed in this manner for a fixed number of times as determined by the counter.

Instruction 10 is the conditional RETURN-FROM-SUBROUTINE instruction. As the name implies, this instruction is used to branch from the subroutine back to the next microinstruction address following the subroutine call. Since this instruction is conditional, the return is performed only if the test is passed. If the test is failed, the next sequential microinstruction is performed. The example in Figure a depicts the use of the conditional RETURN-FROM-SUBROUTINE instruction in both the conditional and the unconditional modes. This example first shows a jump-to-subroutine at instruction location 52 where control is transferred to location 90. At location 93, a conditional RETURN-FROM-SUBROUTINE instruction is performed. If the test is passed, the stack is accessed and the program will transfer to the next instruction at address 53. If the test if failed, the next microinstruction at address 94 will be executed. The program will continue to address

97 where the subroutine is complete. To perform an unconditional RETURN-FROM-SUBROUTINE, the conditional RETURN-FROM-SUBROUTINE instruction is executed unconditionally; the microinstruction at address 97 is programmed to force $\overline{CCEN}$ HIGH, disabling the test and the forced PASS causes an unconditional return.

Instruction 11 is the CONDITIONAL JUMP PIPELINE register address and POP stack instruction. This instruction provides another technique for loop termination and stack maintenance. The example in Figure a shows a loop being performed from address 55 back to address 51. The instructions at location 52, 53, and 54 are all conditional JUMP and POP instructions. At address 52, if the $\overline{CC}$ input is LOW, a branch will be made to address 70 and the stack will be properly maintained via a POP. Should the test fail, the instruction at location 53 (the next sequential instruction) will be executed. Likewise, at address 53, either the instruction at 90 or 54 will be subsequently executed, respective to the test being passed or failed. The instruction at 54 follows the same rules, going to either 80 or 55. An instruction sequence as described here, using the CONDITIONAL JUMP PIPELINE and POP instruction, is very useful when several inputs are being tested and the microprogram is looping waiting for any of the inputs being tested to occur before proceeding to another sequence of instructions. This provides the powerful jump-table programming technique at the firmware level.

Instruction 12 is the LOAD COUNTER AND CONTINUE instruction, which simply enables the counter to be loaded with the value at its parallel inputs. These inputs are normally connected to the pipeline branch address field which (in the architecture being described here) serves to supply either a branch address or a counter value depending upon the microinstruction being executed. There are altogether three ways of loading the counter—the explicit load by this instruction 12; the conditional load included as part of instruction 4; and the use of the $\overline{RDL}$ input along with any instruction. The use of $\overline{RDL}$ with any instruction overrides any counting or decrementation specified in the instruction, calling for a load instead. Its use provides additional microinstruction power, at the expense of one bit of microinstruction width. This instruction 12 is exactly equivalent to the combination of instruction 14 and $\overline{RLD}$ LOW. Its purpose is to provide a simple capability to load the register/counter in those implementations which do not provide microprogrammed control for $\overline{RDL}$.

Instruction 13 is the TEST END-OF-LOOP instruction, which provides the capability of conditionally exiting a loop at the bottom; that is, this is a conditional instruction that will cause the microprogram to loop, via the file, if the test is failed else to continue to the next sequential instruction. The example in Figure a shows the TEST END-OF-LOOP microinstruction at address 56. If the test fails, the microprogram will branch to address 52. Address 52 is on the stack because a PUSH instruction had been executed at address 51. If the test is passed at instruction 56, the loop is terminated and the next sequential microinstruction at address 57 is executed, which also causes the stack to be POPed, thus accomplishing the required stack maintenance.

Instruction 14 is the CONTINUE instruction, which simply causes the microprogram counter to increment so that the next sequential microinstruction is executed. This is the simplest microinstruction of all and should be the default instruction which the firmware requests whenever no other instruction is being executed.

Instruction 15, THREE-WAY BRANCH, is the most complex. It provides for testing of both a data-dependent condition and the counter during one microinstruction and provides for selecting among one of three microinstruction addresses as the next microinstruction to be performed. Like instruction 8, a previous instruction will have loaded a count into the register/counter while pushing a microbranch address onto the stack. Instruction 15 performs a decrement-and-branch-until-zero function similar to intruction 8. The next address is taken from the top of the stack until the count reaches zero; then the next address comes from the pipeline register. The above action continues as long as the test condition fails. If at any execution of instruction 15 the test condition is passed, no branch is taken; the microprogram counter register furnishes the next address. When the loop is ended, either by the count becoming zero or by passing the conditional test, the stack is POPed by decrementing the stack pointer, since interest in the value contained at the top of the stack is then complete.

The application of instruction 15 can enhance performance of a variety of machine-level instructions. For instance, (1) a memory search instruction to be terminated either by finding a desired memory content or by reaching the search limit; (2) variable-field-length arithmetic terminated early upon finding that the content of the portion of the field still unprocessed is all zeroes; (3) key search in a disc controller processing variable length records; (4) normalization of a floating point number.

As one example, consider the case of a memory search instruction. As shown in Figure a, the instruction at microprogram address 63 can be Instruction 4 (PUSH), which will push the value 64 onto the microprogram stack and load the number N, which is one less than the number of memory locations to be searched before giving up. Location 64 contains a microinstruction which fetches the next operand from the memory area to be searched and compares it with the search key. Location 65 contains a microinstruction which tests the result of the comparison and also is a THREE-WAY BRANCH for microprogram control. If no match is found, the test fails and the microprogram goes back to location 64 for the next operand address. When the count becomes zero, the microprogram branches to location 72, which does whatever is necessary if no match is found. If a match occurs on any execution of the THREE-WAY BRANCH at location 65, control falls through to location 66 which handles this case. Whether the instruction ends by finding a match or not, the stack will have been POPed once, removing the value 64 from the top of the stack.

Table 1 Instruction Set

				Fail $\overline{CCEN}$ = LOW and $\overline{CC}$ = HIGH		Pass $\overline{CCEN}$ = HIGH or $\overline{CC}$ = LOW			
I_3-I_0	Mnemonic	Name	Reg/ Cntr Contents	Y	Stack	Y	Stack	Reg/ Cntr	Enable
0	JZ	JUMP ZERO	X	0	CLEAR	0	CLEAR	HOLD	PL
1	CJS	COND JSB PL	X	PC	HOLD	D	PUSH	HOLD	PL
2	JMAP	JUMP MAP	X	D	HOLD	D	HOLD	HOLD	MAP
3	CJP	COND JUMP PL	X	PC	HOLD	D	HOLD	HOLD	PL
4	PUSH	PUSH/COND LD CNTR	X	PC	PUSH	PC	PUSH	Note 1	PL
5	JSRP	COND JSB R/PL	X	R	PUSH	D	PUSH	HOLD	PL
6	CJV	COND JUMP VECTOR	X	PC	HOLD	D	HOLD	HOLD	VECT
7	JRP	COND JUMP R/PL	X	R	HOLD	D	HOLD	HOLD	PL
8	RFCT	REPEAT LOOP, CNTR ≠ 0	≠ 0	F	HOLD	F	HOLD	DEC	PL
			= 0	PC	POP	PC	POP	HOLD	PL
9	RPCT	REPEAT PL, CNTR ≠ 0	≠ 0	D	HOLD	D	HOLD	DEC	PL
			= 0	PC	HOLD	PC	HOLD	HOLD	PL
10	CRTN	COND RTN	X	PC	HOLD	F	POP	HOLD	PL
11	CJPP	COND JUMP PL & POP	X	PC	HOLD	D	POP	HOLD	PL
12	LDCT	LD CNTR & CONTINUE	X	PC	HOLD	PC	HOLD	LOAD	PL
13	LOOP	TEST END LOOP	X	F	HOLD	PC	POP	HOLD	PL
14	CONT	CONTINUE	X	PC	HOLD	PC	HOLD	HOLD	PL
15	TWB	THREE-WAY BRANCH	≠ 0	F	HOLD	PC	POP	DEC	PL
			= 0	D	POP	PC	POP	HOLD	PL

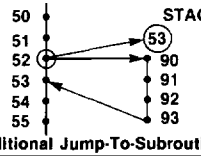
Note 1: If $\overline{CCEN}$ = LOW and $\overline{CC}$ = HIGH, hold else load X = Don't Care

Fig. a Execution Examples

INSTRUCTION 0

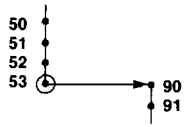
Jump Zero (JZ)

INSTRUCTION 1



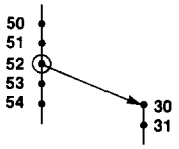
Conditional Jump-To-Subroutine (CJS)

INSTRUCTION 2



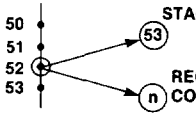
Jump Map (JMAP)

INSTRUCTION 3



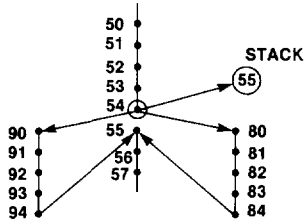
Conditional Jump Pipeline (CJP)

INSTRUCTION 4



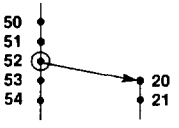
Push/Conditional Load Counter (PUSH)

INSTRUCTION 5



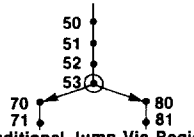
Conditional Jump-To-Subroutine Via Register or Pipeline Register (COND JSB R/PL)

INSTRUCTION 6



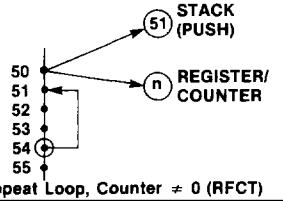
Conditional Jump Vector (CJV)

INSTRUCTION 7



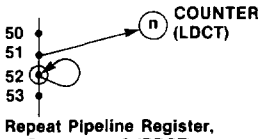
Conditional Jump Via Register or Pipeline Register (JRP)

INSTRUCTION 8



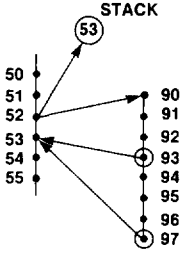
Repeat Loop, Counter ≠ 0 (RFCT)

INSTRUCTION 9



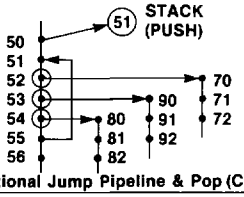
Repeat Pipeline Register, Counter ≠ 0 (RPCT)

INSTRUCTION 10



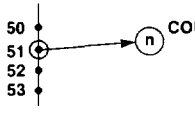
Return From Subroutine (CRTN)

INSTRUCTION 11



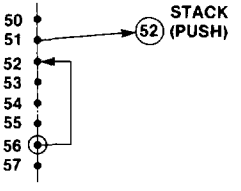
Conditional Jump Pipeline & Pop (CJPP)

INSTRUCTION 12



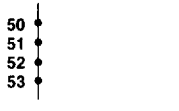
Load Counter and Continue (LDCT)

INSTRUCTION 13



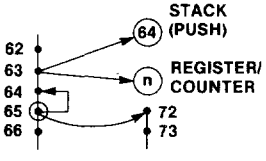
Test End-Of-Loop (LOOP)

INSTRUCTION 14



Continue (CONT)

INSTRUCTION 15



Three-Way Branch (TWB)

DC Characteristics over Operating Temperature Range (unless otherwise specified)

Symbol	Parameter	29F			Units	Conditions
		Min	Typ	Max		
I_{CC}	Power Supply Current		195	295	mA	V_{CC} = Max

AC Characteristics: See Section 3 for waveforms and load configurations

Symbol	Parameter	29F T_A = +25°C V_{CC} = +5.0 V C_L = 50 pF			Military 29F T_A, V_{CC} = Mil C_L = 50 pF		Commercial 29F T_A, V_{CC} = Com C_L = 50 pF		Units
		Min	Typ	Max	Min	Max	Min	Max	
t_{PLH} t_{PHL}	Propagation Delay D_n to Y			26.0 26.0					ns
t_{PLH} t_{PHL}	Propagation Delay I_n to Y			40.0 40.0					ns
t_{PLH} t_{PHL}	Propagation Delay I_n to PL, VECT, MAP			30.0 30.0					ns
t_{PLH} t_{PHL}	Propagation Delay $\overline{CC}$ to Y			35.0 35.0					ns
t_{PLH} t_{PHL}	Propagation Delay $\overline{CCEN}$ to Y			35.0 35.0					ns
t_{PLH} t_{PHL}	Propagation Delay CP to Y			30.0 30.0					ns
t_{PLH} t_{PHL}	Propagation Delay CP (I = 8, 9, 15) to Y			55.0 55.0					ns
t_{PLH} t_{PHL}	Propagation Delay CP to $\overline{FULL}$			40.0 40.0					ns
t_{PLH} t_{PHL}	Propagation Delay CP (I = 8, 9, 15) to $\overline{FULL}$			40.0 40.0					ns
t_{PZH} t_{PZL}	Output Enable Time $\overline{OE}$ to Y			17.0 17.0					ns
t_{PHZ} t_{PLZ}	Output Disable Time $\overline{OE}$ to Y			21.0 21.0					ns

AC Operating Requirements: See Section 3 for waveforms

Symbol	Parameter	29F	Military 29F	Commercial 29F	Units
		$T_A = +25°C$ $V_{CC} = +5.0\ V$	T_A, V_{CC} = Mil	T_A, V_{CC} = Com	
		Min Typ Max	Min Max	Min Max	
$t_s(H)$ $t_s(L)$	Setup Time, HIGH or LOW D (RC) to CP	7.0 7.0			ns
$t_h(H)$ $t_h(L)$	Hold Time, HIGH or LOW D (RC) to CP	0 0			ns
$t_s(H)$ $t_s(L)$	Setup Time, HIGH or LOW D (PC) to CP	9.0 9.0			ns
$t_h(H)$ $t_h(L)$	Hold Time, HIGH or LOW D (PC) to CP	0 0			ns
$t_s(H)$ $t_s(L)$	Setup Time, HIGH or LOW I_0-I_3 to CP	20.0 20.0			ns
$t_h(H)$ $t_h(L)$	Hold Time, HIGH or LOW I_0-I_3 to CP	6.0 0			ns
$t_s(H)$ $t_s(L)$	Setup Time, HIGH or LOW $\overline{CC}$ to CP	21.0 21.0			ns
$t_h(H)$ $t_h(L)$	Hold Time, HIGH or LOW $\overline{CC}$ to CP	0 0			ns
$t_s(H)$ $t_s(L)$	Setup Time, HIGH or LOW $\overline{CCEN}$ to CP	18.0 18.0			ns
$t_h(H)$ $t_h(L)$	Hold Time, HIGH or LOW $\overline{CCEN}$ to CP	0 0			ns
$t_s(H)$ $t_s(L)$	Setup Time, HIGH or LOW $\overline{CI}$ to CP	12.0 12.0			ns
$t_h(H)$ $t_h(L)$	Hold Time, HIGH or LOW $\overline{CI}$ to CP	0 0			ns
$t_s(H)$ $t_s(L)$	Setup Time, HIGH or LOW $\overline{RLD}$ to CP	24.0 24.0			ns
$t_h(H)$ $t_h(L)$	Hold Time, HIGH or LOW $\overline{RLD}$ to CP	0 0			ns
$t_w(H)$ $t_w(L)$	Clock Pulse Width HIGH or LOW	8.0 12.0			ns

Fig. b Switching Waveforms