Am2914

HARDWARE LEVEL INTERRUPTS FOR THE 2900 FAMILY

by Donnamaie E. White

Interrupt handling at the firmware or the software level is satisfactory for CPUs, emulators, and other systems where the interrupts are to be recognized under programmer control. These "soft" interrupts are not adequate when the response must be immediate. When "realtime" response is desired then hardware level interrupts are necessary.

There are a number of solutions possible using the Am2910 microprogram sequencer and the Am2914 priority interrupt controller. Three of these solutions appeared in a paper by Vernon Coleman ("Microprogram Interrupts: When and How"), segments of which have appeared in various articles. This application note will discuss those three techniques in detail and clarify their operation.

Figure 1 shows a "typical" microprogram control interrupt scheme. The sequencer is the Am2910 and the three address sources for the Am2910 are the pipeline branch address field (Am29774/5s), the memory map (labeled instruction map on the figure), and the vector map. The Am2910 is shown connected to all of the address source output enables. The interrupt request line of the Am2914 is the only test input available to the Am2910 in this figure and therefore the traditional conditional multiplexer is absent. There are four devices connected to the interrupt request inputs of the Am2914 (prioritized). No ALU is shown.

FIGURE 1.
TYPICAL MICROPROGRAM CONTROL INTERRUPT SCHEME

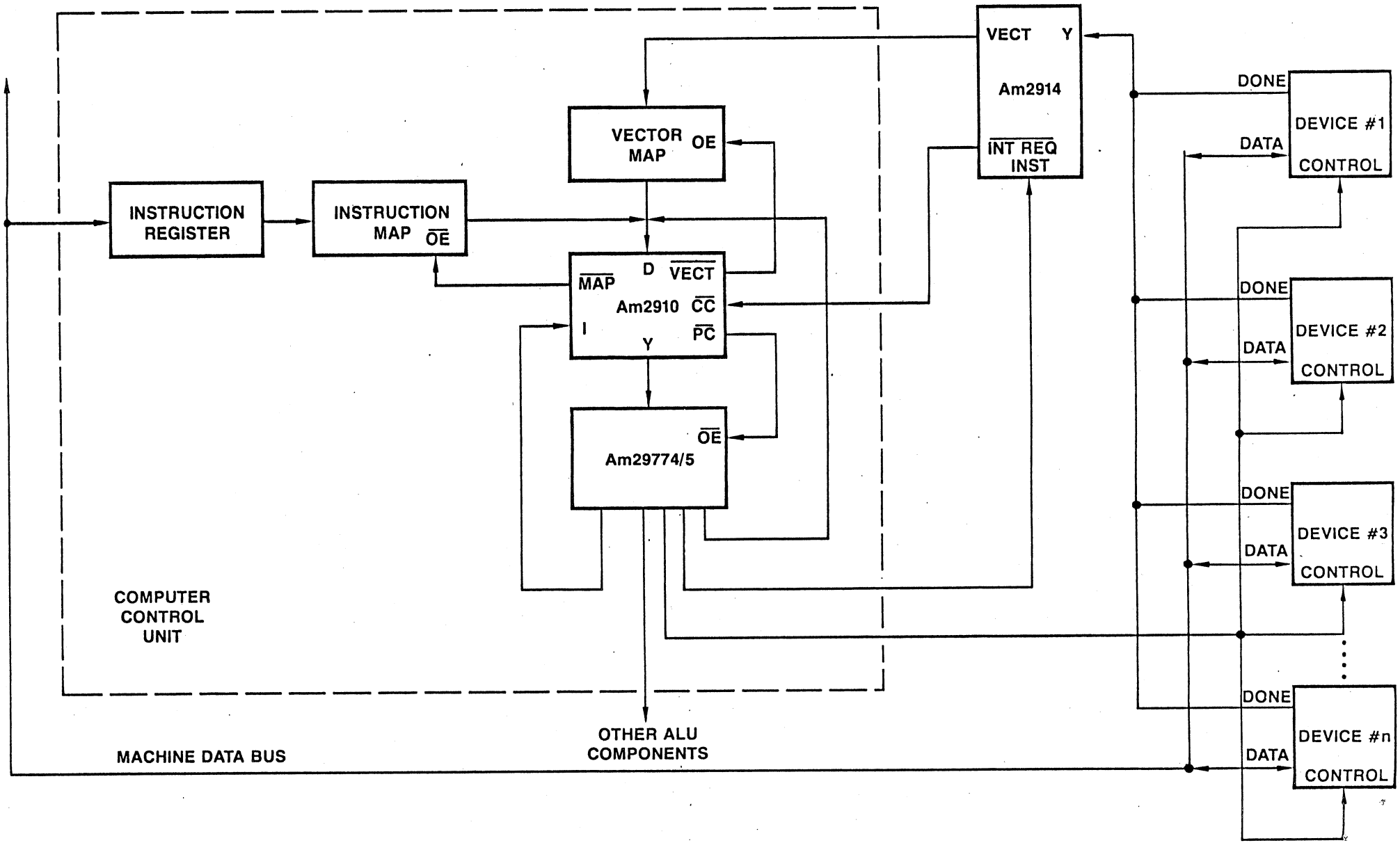


Figure 2 shows the overview flowchart for the traditional placement of interrupt testing using CJV (and not CJS). The number of microinstructions between interrupt tests is various depending on the microroutine being executed. It is satisfactory for a CPU.

Figure 3 shows the desirable flowchart where a test for interrupt occurrence is made every other microcycle. This doubles the microcode required to execute any sequence and halves the throughput and is not acceptable for CPUs or controllers.

Figure 4. Microword format for a typical microprogram control interrupt scheme (firmware level).

addr <- SEQUENCE CONTROL -> <--- INTERRUPT ----> <- OUTPUT ENABLES ->

	Am2910 INSTR	COND MUX	BR ADDR/ COUNTER	Am2914 INSTR	Am2914 $\overline{\text{Ien}}$	Am2914 $\overline{\text{INT DIS}}$	$\overline{OE}_{PL}$	$\overline{OE}_{MAP}$	$\overline{OE}_{VECT}$	OTHER...
	4	3	12	4	1	1	1	1	1	● ● ●
i:	CJV	ANY	#	READVCTR	EN	EN	DIS	DIS	EN	● ● ●

Figure 4 above shows the microword required for the case diagrammed in figure 2. Remember that when the interrupts are tested at only one location (within the common code segment) then a return address need not be saved. This means that a subroutine is not required and also means that nesting of interrupts is not permissible. CJV, the conditional jump vector instruction of the Am2910 is adequate and the three address sources for the Am2910 may be controlled via the output enables provided by the Am2910 or via the microword itself as shown. Remember the speed versus microword size tradeoff.

FIGURE 2.
EXECUTING INTERRUPTS ON INSTRUCTION BOUNDARIES

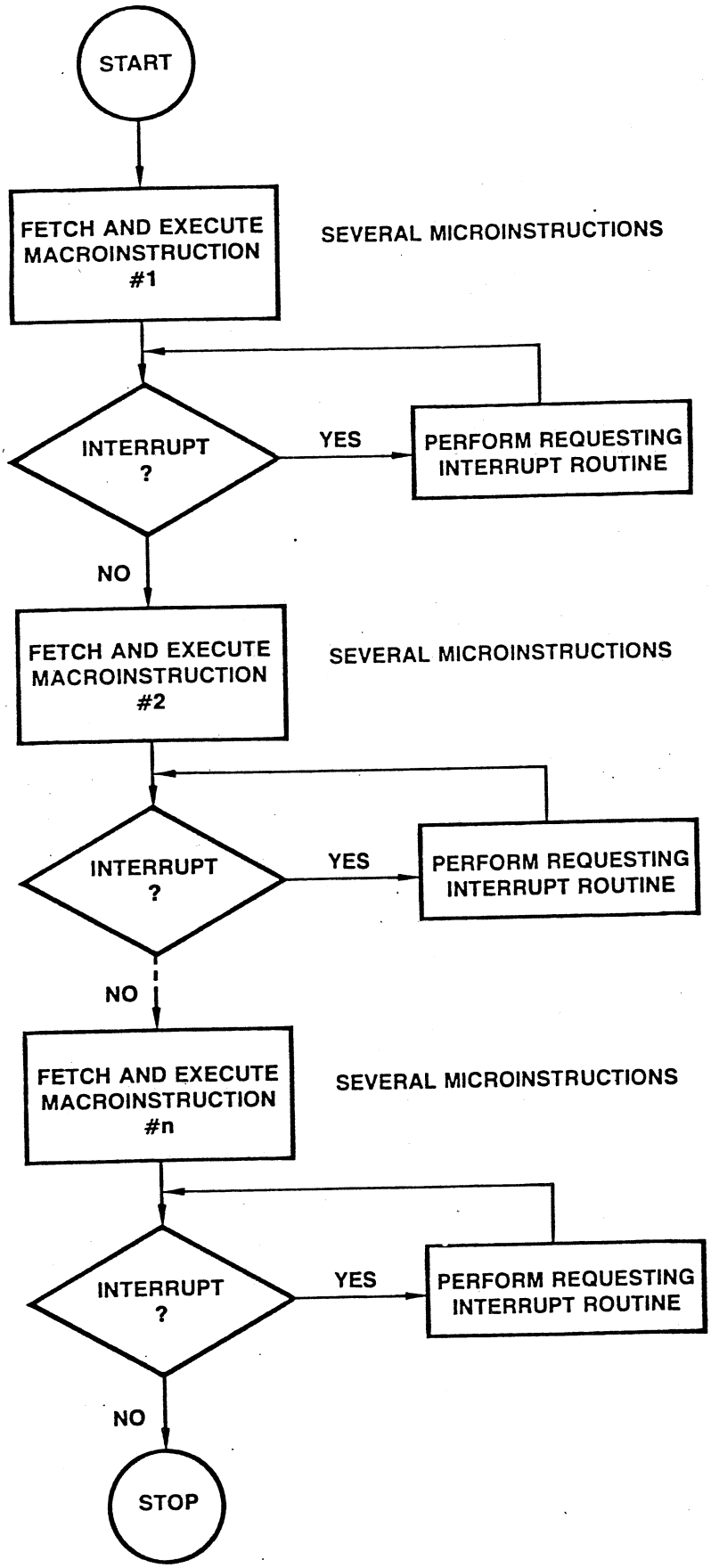
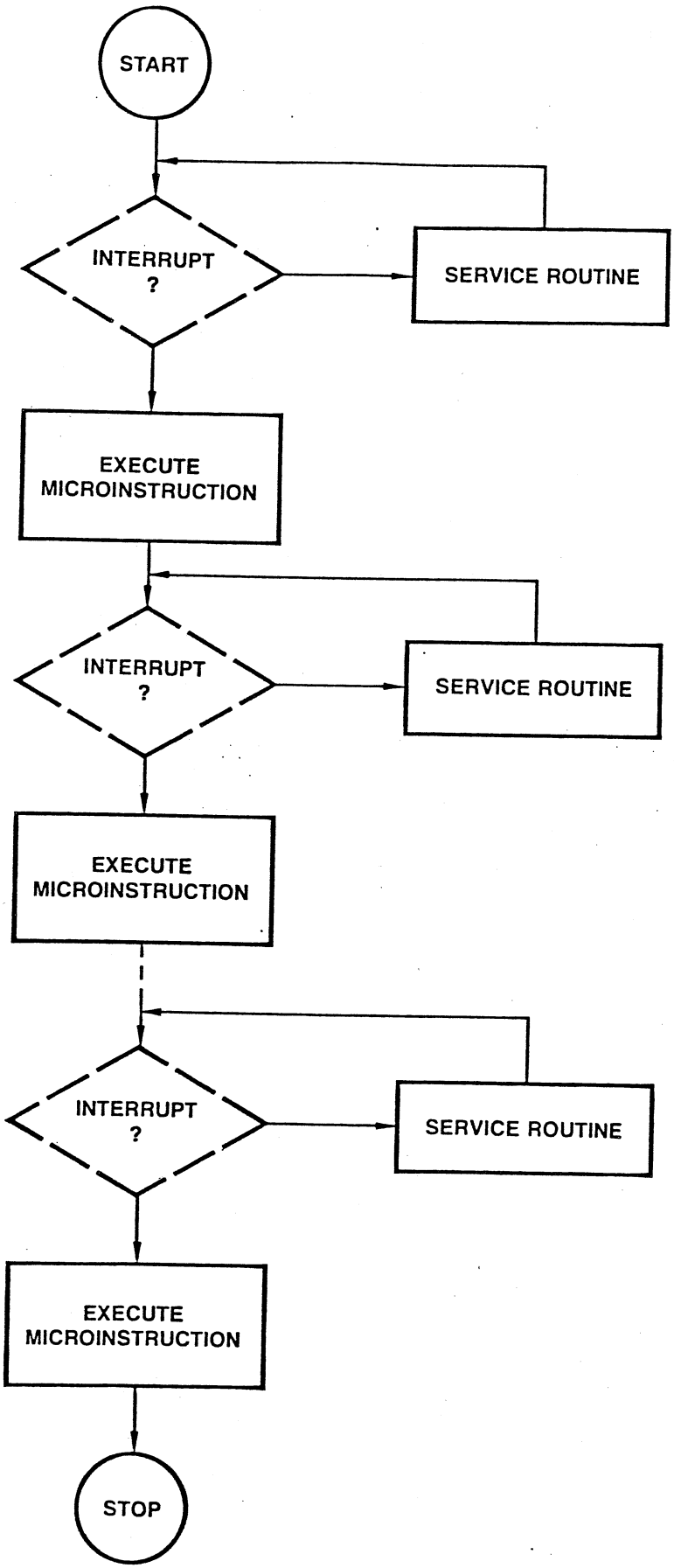


FIGURE 3.
TRANSPARENT TESTING FOR INTERRUPTS



Am2914

Figure 5. Microword for "realtime" microprogram interrupt control.

addr <- SEQUENCE CONTROL -> <--- INTERRUPT ----> <- OUTPUT ENABLES ->

	Am2910 INSTR	COND MUX	BR ADDR/ COUNTER	Am2914 INSTR	Am2914 $\overline{Ien}$	Am2914 INT DIS	$\overline{OE}_{PL}$	$\overline{OE}_{MAP}$	$\overline{OE}_{VECT}$	OTHER...
	4	3	12	4	1	1	1	1	1	•••
i:	CJS	ANY	#	READVCTR	EN	EN	DIS	DIS	EN	•••

Figure 5 shows the microword for the case diagrammed in Figure 3. This microinstruction would have to be inserted in the microprogram such that it would be executed every other microcycle. Obviously, it is desirable to find a more practical solution (in terms of throughput) than what can be obtained via a firmware solution.

HARDWARE SOLUTIONS

The three hardware solutions which will be discussed are:

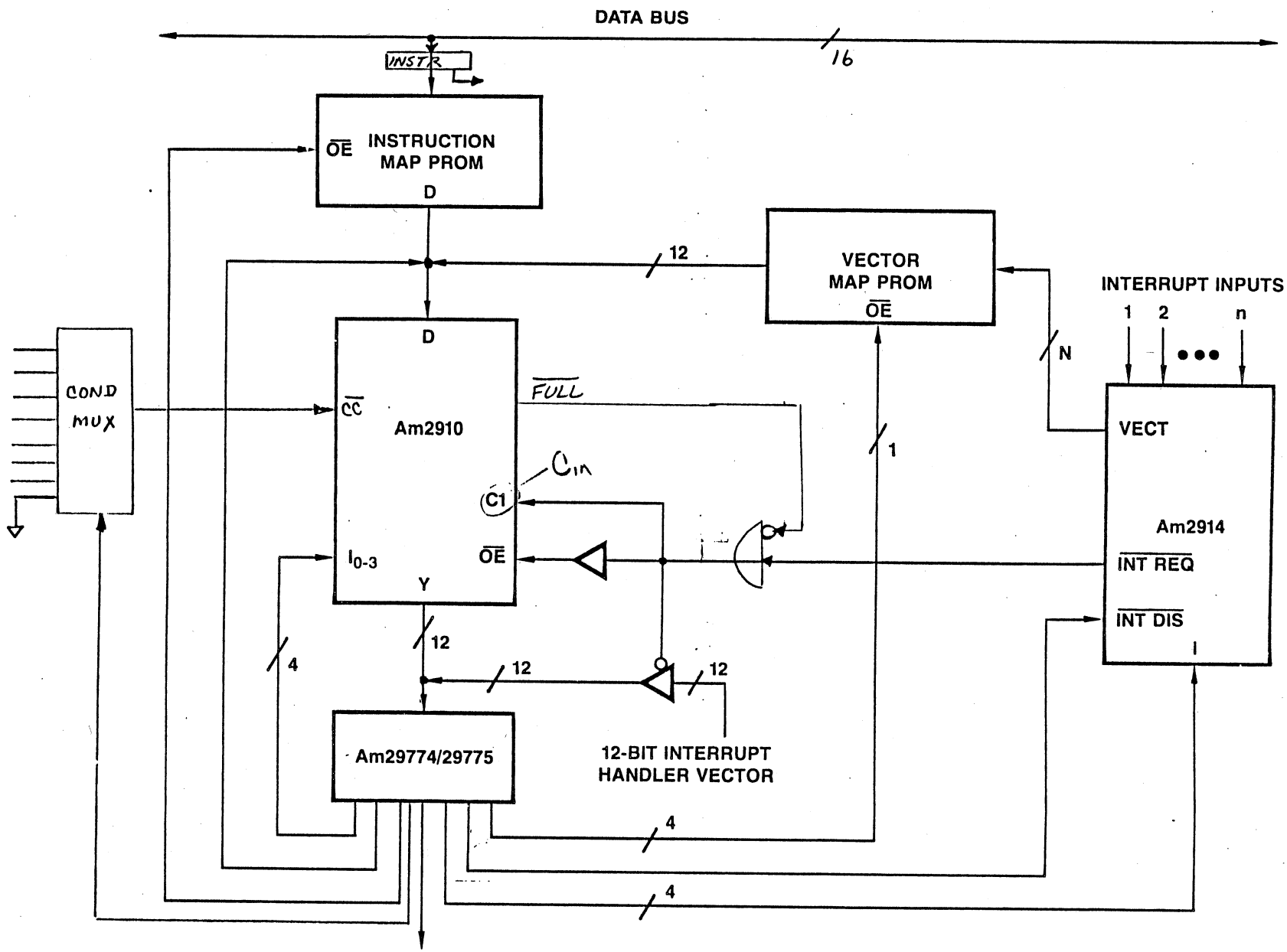
1) a single microprogram controller realtime interrupt scheme
2) external storage for the microprogram controller
3) multiple microprogram controllers with realtime interrupt handling

SINGLE MICROPROGRAM CONTROLLER REALTIME INTERRUPT SCHEME

The basic hardware configuration for this scheme is given in figure 6. The interrupt request line ("ANY") is connected to the output enable pin of the Am2910, the carry-in pin of the incrementer on-board the Am2910, and the enable to the hardwired interrupt vector. (This is a modification to the drawing shown in the paper by Vernon Coleman.) The conditional multiplexer is shown connected to the test input on the Am2910 and all other tests would be made via firmware.

FIGURE 4. 6-7

SINGLE SEQUENCER INTERRUPT SCHEME



Am2914

On the occurrence of an interrupt, the Am2910 operation is suspended at the point in the microprogram at which it was executing by pulling the carry-in pin LOW (which prevents the incrementer from incrementing, i.e., the microprogram register will not advance) and by pulling the output enable high, disabling the Yi outputs of the Am2910 from the microprogram control memory address lines (Am29774/5s).

Notice that the $\overline{\text{FULL}}$ pin of the Am2910 is used to disable the interrupt request from reaching the Am2910.

INTERRUPT DURING SEQUENTIAL EXECUTION

The microinstruction which is in process (which is resident in the microinstruction or pipeline register) is completed while the contents of the microprogram counter (uPC), the address of the next instruction in sequence, is held constant. On the next clock rising edge the uPC register will receive the same contents that it had held prior to the clock because the carry-in pin is LOW. Since the Am2910 outputs are disabled, the tristated outputs are pulled to the value of the enabled interrupt handler vector.

INTERRUPT WHEN BRANCHING

If an interrupt occurs when a branch address is being fetched, the process is the same with the following exception. The uPC register will contain the non-incremented branch address after the rising edge of the next clock cycle. If the branch was a branch to a subroutine, and the stack would have been full after the rising edge of the next clock, there is a problem! If the main body of code and the interrupt routines themselves contain no imbedded subroutines, then this deficiency will not matter.

INTERRUPT HANDLER

The interrupt handler drives a specific address onto the address lines of the microprogram control memory. Note that the same address is used for every interrupt. The instruction that would be placed at that address is shown in figure 7.

Figure 7. Interrupt handler microinstruction.

addr	<- SEQUENCE CONTROL ->			<--- INTERRUPT ---->			<- OUTPUT ENABLES ->			
	Am2910 INSTR	COND MUX	BR ADDR/ COUNTER	Am2914 INSTR	Am2914 Ien	Am2914 INT DIS	$\overline{OE}_{PL}$	$\overline{OE}_{MAP}$	$\overline{OE}_{VECT}$	OTHER...
	4	3	12	4	1	1	1	1	1	•••
hdlr:	CJS	PASS	#	READVCTR	EN	DIS	DIS	DIS	EN	•••

The execution of the handler CJS microinstruction is carried out with a disable sent to the Am2914 to prevent any other interrupt from destroying the handler. The first microinstruction in every interrupt routine must also show the Am2914 interrupts disabled while it clears the last vector read (set by READ VECTOR). The second microinstruction then enables the interrupt request. These instructions are shown in figure 8. The last microinstruction in the interrupt routine is a return to the interrupted sequence, also executed with interrupts disabled. Remember that there is a limit to the depth that these interrupts may nest themselves and that is controlled by the depth of the stack on the Am2910 (five).

Am2914

Figure 8. Interrupt routine microinstructions.

addr <- SEQUENCE CONTROL -> <--- INTERRUPT ----> <- OUTPUT ENABLES ->

	Am2910 INSTR	COND MUX	BR ADDR/ COUNTER	Am2914 INSTR	Am2914 $\overline{Ien}$	Am2914 $\overline{INT\ DIS}$	$\overline{OE}_{PL}$	$\overline{OE}_{MAP}$	$\overline{OE}_{VECT}$	OTHER...
	4	3	12	4	1	1	1	1	1	● ● ●
i:	CONT	#	#	CLRLSTRD	EN	DIS	EN	DIS	DIS	● ● ●
i+1:	CONT	#	#	ENABLINTR	EN	EN	EN	DIS	DIS	● ● ●
i+m:	CRTN	PASS	#	#	DIS	DIS	EN	DIS	DIS	● ● ●

CLRLSTRD = CLEAR LAST VECTOR READ
ENABLINTR = ENABLE INTERRUPTS (IF DISABLED PREVIOUSLY)

PROBLEMS

The first problem has already been mentioned, that is, if a branch to a subroutine is in process, there is no way to know whether or not the Am2910 stack is full. This can be resolved by always executing CJS with the interrupts disabled.

(The original design did not make use of the $\overline{FULL}$ pin.)

The second problem with this design is the failure to provide storage for the register/counter. This means that either all interrupts must be disabled during loops which make use of the counter, or code segments which use the register to store a later-referenced address, or no interrupt routine may make use of the register/counter. If no interrupt routine makes use of the register/counter, then there is no problem with lengthy stretches of execution during which interrupts are ignored.

This technique requires a minimal amount of SSI logic to implement and uses only one or two "overhead" instructions.

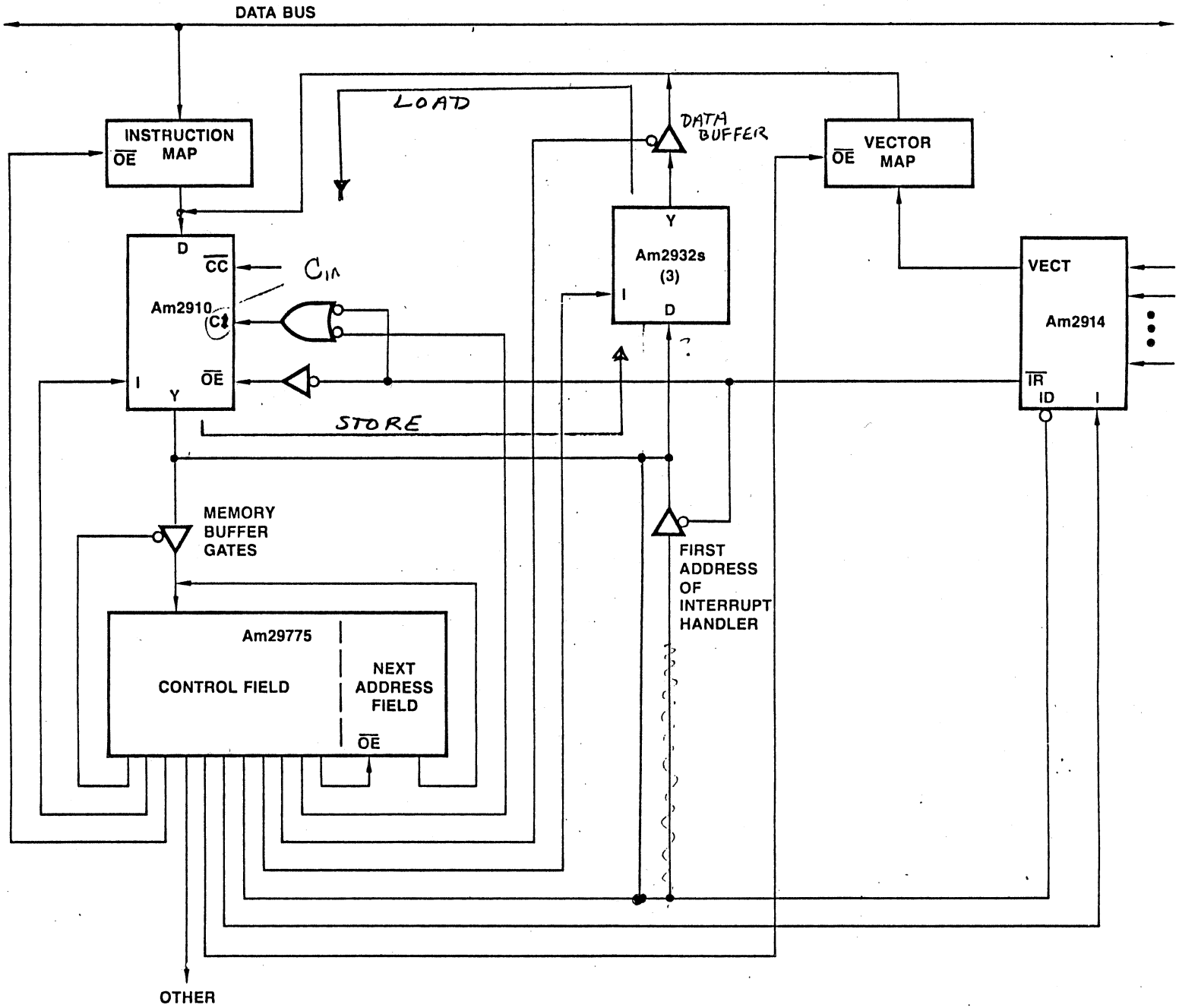
EXTERNAL STORAGE FOR THE Am2910

The second realtime interrupt approach is shown in figure 9. This implementation provides an array of three Am2932 program control units (used as a LIFO stack). The Am2932 instruction lines I2 and I3 are grounded. This structure enables an interrupt to suspend the Am2910 as in the previous example. The additional control here is to allow the carry-in pin of the Am2910 to be re-enabled while keeping the Yi outputs disabled.

On interrupt, the Am2910 is suspended as before. The difference is that the first microinstruction executed (which was the handler step) now enables the next address field in the microprogram control memory. The address which is hardwired is the address of an instruction causing control of the system to be turned over to a portion of the control memory which is essentially a state machine. This state machine proceeds to unload the Am2910 local control storage into the Am2932 (onto a stack) via repeated PUSHs.

Note that the Am2910 is operating but is not allowed to alter any of its internal storage (other than to offload it) and is prevented, via the memory buffer gates, from reaching the address lines of the control memory.

FIGURE 5.
EXTERNALLY STORED SEQUENCER
STATE INTERRUPT SCHEME



The first seven steps of the handler (the state machine) implement this activity. The eighth step, shown in figure 10, is a jump to an individual interrupt routine. At this point the Am2910 resumes normal operation and the next address field (used only by the state machines) is disabled. The interrupt routine clears the interrupt and re-enables the Am2914. The last step of all interrupt routines is a jump back to the handler (CJP since the address is fixed and known).

The last seven microinstructions of the handler are also controlled by a state machine and this time the reverse operation is performed. The Am2910 is again restricted from reaching the control memory address lines while the stack on the Am2932s is POPped and the original state of the Am2910 is restored.

Figure 10 lays out the partial microword for this part of the system and provided the microwords required for the two state machines and the initial steps of any interrupt routine. Notice that this implementation requires a much wider microinstruction and additional hardware over the previous solution.

	addr	SEQUENCE CONTROL: 2910 INSTR	COND MUX	BR ADDR COUNTER	INTERRUPT HANDLING: 2914 INSTR	$\overline{I}_{EN}$	2914 INTR DIS	BRANCH ENABLES: $\overline{OE}_{PL}$	$\overline{OE}_{MAP}$	$\overline{OE}_{VECT}$	INTERRUPT SUPPORT: MEM BUFFER	CARRY ENABLE	DATA BUS BUFFER	ADR EN	NXT ADR	2932 INST	COMMENT
handler	h	CONT	#	#	#	DIS	DIS	DIS	DIS	DIS	DIS	DIS	DIS	EN	h+1	PSHD	SAVE μPC
	h+1	CRTN	PASS	#	#	DIS	DIS	DIS	DIS	DIS	DIS	DIS	DIS	EN	h+2	PSHD	SAVE STACK
	h+2	CRTN	PASS	#	#	DIS	DIS	DIS	DIS	DIS	DIS	DIS	DIS	EN	h+3	PSHD	
	h+3	CRTN	PASS	#	#	DIS	DIS	DIS	DIS	DIS	DIS	DIS	DIS	EN	h+4	PSHD	
	h+4	CRTN	PASS	#	#	DIS	DIS	DIS	DIS	DIS	DIS	DIS	DIS	EN	h+5	PSHD	
	h+5	CRTN	PASS	#	#	DIS	DIS	DIS	DIS	DIS	DIS	DIS	DIS	EN	h+6	PSHD	
	h+6	CJRP	FAIL	#	#	DIS	DIS	DIS	DIS	DIS	DIS	DIS	DIS	EN	h+7	PSHD	SAVE R
	h+7	CJV	PASS	#	RDVCTR	EN	DIS	DIS	DIS	EN	EN	EN	DIS	DIS	#	SUSPEND	BRANCH VECTOR
INTERRUPT ROUTINE	i	CONT	#	#	CLRINT	EN	DIS	EN	DIS	DIS	EN	EN	DIS	DIS	#	SUSPEND	CLEAR INTERRUPT
	i+1	CONT	#	#	ENINTR	EN	EN	EN	DIS	DIS	EN	EN	DIS	DIS	#	SUSPEND	ENABLE INTER.
	⋮																
	i+N	CJP	PASS	h+8	#	DIS	DIS	EN	DIS	DIS	EN	EN	DIS	DIS	#	SUSPEND	RETURN TO HANDLER
handler	h+8	LDCT	#	#	#	DIS	DIS	DIS	DIS	DIS	DIS	DIS	EN	EN	h+9	POPS	LOAD R
	h+9	CJS	PASS	|	|	|	|	|	|	|	|	|	EN	EN	h+A	POPS	LOAD STACK
	h+A	CJS	PASS	|	|	|	|	|	|	|	|	|	EN	EN	h+B	POPS	
	h+B	CJS	PASS	|	|	|	|	|	|	|	|	|	EN	EN	h+C	POPS	
	h+C	CJS	PASS	|	|	|	|	|	|	|	|	|	EN	EN	h+D	POPS	
	h+D	CJS	PASS			DIS	DIS				DIS	EN	EN	EN	h+E	POPS	
	h+E	CJP	PASS	#	ENINTR	EN	EN	DIS	DIS	DIS	EN	EN	EN	DIS	#	POPS	LOAD μPC

Figure 10

Microinstructions Supporting the LIFO interrupt scheme

PROBLEMS

The problems with this suggested solution are obvious (or should be if you have taken ED2910, ED2900A or ED2900B).

First, there is an incredibly high overhead (fifteen microcycles per interrupt excluding the interrupt branch and the return microinstructions).

Second, if the Am2910 stack is not full, then it is mishandled. There will always be five POPs even if the stack is empty! There will always be five PUSHs to restore the empty stack, which now thinks that it is full! The solution to this difficulty is to only allow interrupts to occur when the stack is full (which is nonsense) or to add yet more hardware in the form of an external up/down counter whch would keep track of the actual size of the stack. The state machine would now have to incorporate conditional instructions (as per Wilke's paper) such that only valid stack items are saved and restored. This increases the complexity of the solution, the size of the control memory (depth and width) and the related debug, documentation, and other human-comprehension problems.

A compromise solution would be to disable interrupts when the stack is in use and only save the microprogram counter and the register/counter. This would bring us back to approximately the system of figure 6.

Third, the interrupts would have to be disabled during the long overhead steps required for each interrupt.

Fourth, if interrupts are allowed to nest, then the stack on the Am2932 would be exceeded on the third interrupt. The stack on the Am2932 is 17 deep. The Am2932 does have a $\overline{\text{FULL}}$ pin and it could be used to disable interrupts. This leaves us again with long stretches of time when interrupts are not serviced in the manner we specified as desirable.

The conclusion of this author is that this technique is not acceptable.

PARALLEL SEQUENCERS

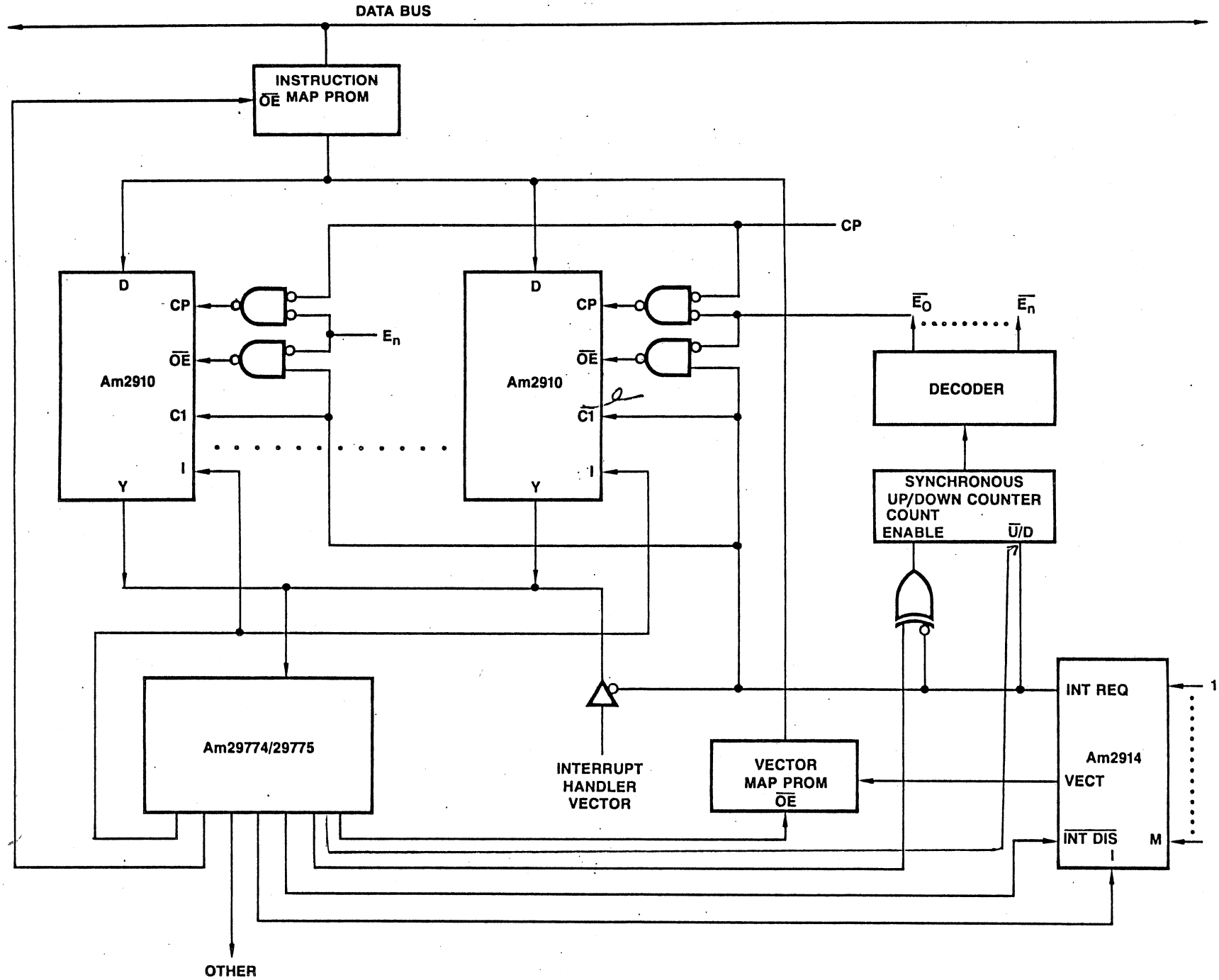
The previously described approach is awkward, conceptually complex, potentially dangerous, and therefore is not recommended except as a mental exercise (to see if you really do understand the parts involved)! A more interesting approach is to "stack switch" between two or more Am2910s connected in parallel. The operational Am2910 would be selected via a synchronous up/down counter and decoder, as shown in figure 11.

The hardware required would be the extra Am2910s (watch the heat/power figures on the board), SSI logic to control both the output enables of the Yi lines of each Am2910 and the clock input pins of each Am2910, the synchronous counter, sized for the application, and a decoder, also sized. A hardwired interrupt handler vector is required as in the other two cases.

The design requires a little more than is shown. The interrupts would have to be disabled when the last Am2910 is active and re-enabled when it becomes available. The number of Am2910s chosen should reflect how deep the interrupts could typically nest and not how deep they could nest in the worst case (unless the application is extremely demanding). The use of a typical depth would be a compromise between cost and performance. If interrupts are not allowed to nest but are handled only one at a time, then interrupts would be disabled any time an interrupt is being serviced and only two Am2910s would be necessary. Two to four is probably an adequate solution, providing fast response for the first one, two, or three to arrive.

F IRE 6.12

PARALLEL SEQUENCER INTERRUPT CONFIGURATION



Am2914

The counter is used to control the active Am2910 as follows. When the counter increments, the decoder generates a different active enable Ei control which in turn causes one Am2910 (the one which was active) to be suspended as is and the next Am2910 in line to be activated. As the interrupt routine ends, the last step is to decrement the counter, which allows the system to return to its previous condition.

The carry-out from the counter can be used to disable the interrupts on the activation of the last Am2910.

Figure 12. Microinstructions to support stack switching.

addr <- SEQUENCE CONTROL -> <--- INTERRUPT ----> <- OUTPUT ENABLES ->

	Am2910 INSTR	COND MUX	BR ADDR/ COUNTER	Am2914 INSTR	Am2914 $\overline{Ien}$	Am2914 INT DIS	$\overline{OE}_{PL}$	$\overline{OE}_{MAP}$	$\overline{OE}_{VECT}$	COUNTER ENABL	OTHER...
	4	3	12	4	1	1	1	1	1		● ● ●
HARDWIRED INSTRUCTION:											
hdlr:	CJS	PASS	#	READVCTR	EN	DIS	DIS	DIS	EN	DIS	● ● ●
ROUTINE:											
i:	CONT	#	#	CLRLSTRD	EN	DIS	EN	DIS	DIS	DIS	● ● ●
i+1:	CONT	#	#	ENINTR	EN	EN	EN	DIS	DIS	EN	● ● ●
i+n:	CRTN	PASS	#	#	DIS	DIS	EN	DIS	DIS	DECR	● ● ●

Am2914

Am29112

The 2900 Family has two basic application areas, computers and controllers. The controllers have a number of requirements that the computers do not have and visa versa. For this reason, there is a new part of the family being created. The first of these will be the Am29116 16-bit controller ALU which handles bit operations, serial I/O, CRC, and provides less CPU and more controller application features. A new sequencer will be developed later on for this family which will be designed specifically to handle realtime interrupts and which will solve the problem with a minimum of hardware. (Sort of a super Am2910). Watch for its announcement.

```
;
; MISC ENABLES
;
EN:             EQU     B#0     ; ACTIVE LOW ENABLE
DIS:            EQU     B#1     ; ACTIVE LOW DISABLE
;
;
;
; Am2932 INSTRUCTION SET
;
RESET:          EQU     H#0     ; RESET
SUSPEND:        EQU     H#1     ; SUSPEND
PUSHD:          EQU     H#2     ; PUSH D
POPS:           EQU     H#3     ; POP S
;
;
```

Am2914

```
        TITLE  PARTIAL .DEF FILE
        WORD 1                                  ; DUMMY INSERT
;
;
; Am2910/Am29811 INSTRUCTIONS
;
JZ:             EQU     H#0     ; JUMP ZERO (RESET)
CJS:            EQU     H#1     ; CONDITIONAL JUMP SUBROUTINE
;
CJP:            EQU     H#3     ; CONDITIONAL JUMP PIPELINE
;
CRTN:           EQU     H#A     ; CONDITIONAL RETURN
;
CONT:           EQU     H#E     ; CONTINUE
;
;
; CONDITIONAL MULTIPLEXER
;
ANY:            EQU     H#0     ; TEST FOR ANY INTERRUPT
                                ; ACTIVE
PASS:           EQU     H#1     ; GROUNDED INPUT = TRUE
FAIL:           EQU     H#9     ; INVERTED GRND LINE (FALSE)
;
;
; Am2914 INSTRUCTION SET
;
MSTRCLR:        EQU     H#0     ; MASTER CLEAR
;
CLRLSTRD:       EQU     H#4     ; CLEAR INTERRUPT - LAST READ
READVCTR:       EQU     H#5     ; READ VECTOR
;
DISABLINT:      EQU     H#D     ; DISABLE INTERRUPT REQUEST
;
ENABLINTR:      EQU     H#F     ; ENABLE INTERRUPT REQUEST
;
```

REFERENCES

1. Am2914 Data Sheet, The Am2900 Family Data Book, pp. 2-158 - 165, 1979.

2. --. "A Microprogrammable, Bipolar, LSI Interrupt Structure Using the Am2914", Ibid., pp. 2-166 - 176.

3. --. "Am2914 Priority Interrupt Encoder Detailed Logic Description", Ibid., pp. 2-177 - 182.

4. Vernon Coleman. "Microprogram Interrupts: When and How", (?).

5. Vern Coleman. "Implementing Interrupts for Bit-slice Procesors", Electronics, April 24, 1980, pp. 159-160.

6. Donnamaie E. White. Bit-Slice Design: Controllers and ALUs. Garland STPM Press, 1981.

7. John Mick and Jim Brick. Bit-Slice Microprocessor Design. Chapter 6, "Interrupt", McGraw-Hill, 1980. (Formerly the Application note series "How to build a microcomputer").