Outline

- Alternative controller FSM implementation approaches based on:
  - Classical Moore and Mealy machines
  - Time state: Divide and Counter
  - Jump counters
  - *Microprogramming (ROM) based approaches*
    - » *branch sequencers*
    - » *horizontal microcode*
    - » *vertical microcode*

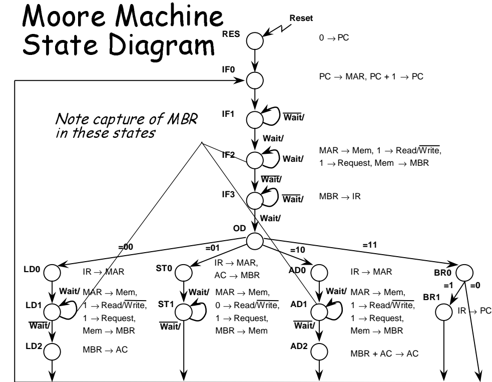
Alternative Ways to Implement Processor FSMs

- "Random Logic" based on Moore and Mealy Design
  - Classical Finite State Machine Design
- Divide and Conquer Approach: Time-State Method
  - Partition FSM into multiple communicating FSMs
- Exploit MSI Functionality: Jump Counters
  - Counters, Multiplexers, Decoders
- Microprogramming: ROM-based methods
  - Direct encoding of next states and outputs

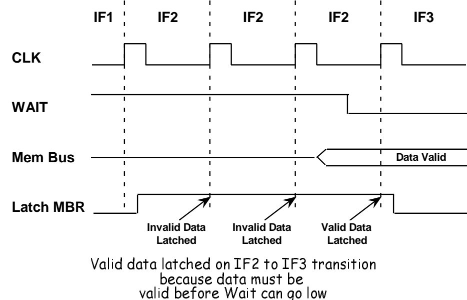
Random Logic

- Perhaps poor choice of terms for "classical" FSMs
- Contrast with structured logic: PAL/PLA, FPGA, ROM
- Could just as easily construct Moore and Mealy machines with these components

Moore Machine State Diagram

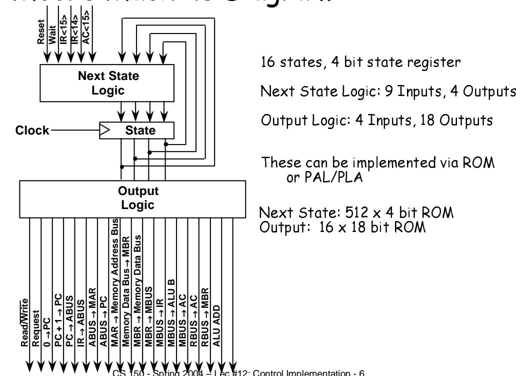


Memory-Register Interface Timing



Valid data latched on IF2 to IF3 transition because data must be valid before Wait can go low

Moore Machine Diagram



16 states, 4 bit state register

Next State Logic: 9 Inputs, 4 Outputs

Output Logic: 4 Inputs, 18 Outputs

These can be implemented via ROM or PAL/PLA

Next State: 512 x 4 bit ROM
Output: 16 x 18 bit ROM

Moore Machine State Table

Reset	Wait	IR<15>	IR<14>	AC<15>	Current State	Next State	Register Transfer Ops
1	X	X	X	X	X	RES (0000)	
0	X	X	X	X	RES (0000)	IF0 (0001)	$0 \rightarrow PC$
0	X	X	X	X	IF0 (0001)	IF1 (0001)	$PC \rightarrow MAR$, $PC + 1 \rightarrow PC$
0	0	X	X	X	IF1 (0010)	IF1 (0010)	
0	1	X	X	X	IF1 (0010)	IF2 (0011)	
0	1	X	X	X	IF2 (0011)	IF2 (0011)	$MAR \rightarrow Mem$, Read,
0	0	X	X	X	IF2 (0011)	IF3 (0100)	Request, $Mem \rightarrow MBR$
0	0	X	X	X	IF3 (0100)	IF3 (0100)	$MBR \rightarrow IR$
0	1	X	X	X	IF3 (0100)	OD (0101)	
0	X	0	0	X	OD (0101)	LD0 (0110)	
0	X	0	1	X	OD (0101)	ST0 (1001)	
0	X	1	0	X	OD (0101)	AD0 (1011)	
0	X	1	1	X	OD (0101)	BR0 (1110)	

Moore Machine State Table

Reset	Wait	IR<15>	IR<14>	AC<15>	Current State	Next State	Register Transfer Ops
0	X	X	X	X	LD0 (0110)	LD1 (0111)	$IR \rightarrow MAR$
0	1	X	X	X	LD1 (0111)	LD1 (0111)	$MAR \rightarrow Mem$, Read,
0	0	X	X	X	LD1 (0111)	LD2 (1000)	Request, $Mem \rightarrow MBR$
0	X	X	X	X	LD2 (1000)	IF0 (0001)	$MBR \rightarrow AC$
0	X	X	X	X	ST0 (1001)	ST1 (1010)	$IR \rightarrow MAR$, $AC \rightarrow MBR$
0	1	X	X	X	ST1 (1010)	ST1 (1010)	$MAR \rightarrow Mem$, Write,
0	0	X	X	X	ST1 (1010)	IF0 (0001)	Request, $MBR \rightarrow Mem$
0	X	X	X	X	AD0 (1011)	AD1 (1100)	$IR \rightarrow MAR$
0	1	X	X	X	AD1 (1100)	AD1 (1100)	$MAR \rightarrow Mem$, Read,
0	0	X	X	X	AD1 (1100)	AD2 (1101)	Request, $Mem \rightarrow MBR$
0	X	X	X	X	AD2 (1101)	IF0 (0001)	$MBR + AC \rightarrow AC$
0	X	X	X	0	BR0 (1110)	IF0 (0001)	
0	X	X	X	1	BR0 (1110)	BR1 (1111)	
0	X	X	X	X	BR1 (1111)	IF0 (0001)	$IR \rightarrow PC$

Moore Machine State Transition Table

- Observations:
  - Extensive use of Don't Cares
  - Inputs used only in a small number of state e.g., AC<15> examined only in BR0 state IR<15:14> examined only in OD state
- Some outputs always asserted in a group
- ROM-based implementations cannot take advantage of don't cares
- However, ROM-based implementation can skip state assignment step

Moore Machine Implementation

Assume PAL/PLA implementation style

First idea: run ESPRESSO with naive state assignment

```
.i 9
.o 4
.ilb reset wait ir15 ir14 ac15 q3 q2 q1 q0
.ob p3 p2 p1 p0
.p 26
1----   ----    0000
0----   0001  0001
00---  0010  0010
01---  0010  0011
01---  0011  0011
00---  0011  0100
00---  0100  0100
01---  0100  0101
0-00-  0101 0110
0-01-  0101 1001
0-10-  0101 1011
0-11-  0101 1110
0----   0110  0111
01---  0111 0111
00---  0111 1000
0----  1000  0001
0----   1001 1010
01---  1010 1010
00---  1010  0001
0----  1011  1100
01---  1100 1100
00---  1100 1101
0----   1101  0001
0---0  1110  0001
0---1  1110 1111
0----   1111  0001
.e
```

21 product terms

Compare with 512 product terms in ROM implementation!

```
.i 9
.o 4
.ilb reset wait ir15 ir14 ac15 q3 q2 q1 q0
.ob p3 p2 p1 p0
.p 21
0-00-0101  0110
0-01-0101  1001
0-11-0101  1110
0-10-0101  1011
01---1010  1010
00---0111  1000
00----011  0100
0----1000  0001
0---11110  1110
01---011-  0100
0----0001  0001
01---01-0  0001
0----1001  1010
0----1011  1100
00---1--0  0001
0----1100  1100
0----0-10  0010
0-----110  0001
0----11-1  0001
0----01-0  0100
01---0-1-  0011
.e
```


Moore Machine Implementation

NOVA assignment does better

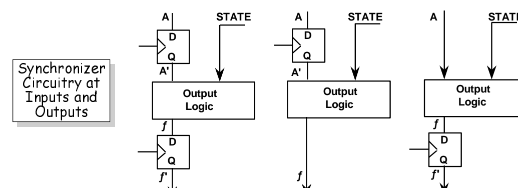
NOVA State Assignment SUMMARY

onehot_products = 22
best_products = 18 — **18 product terms improves on 21!**
best_size = 414

states[0]:IF0 Best code: 0000
states[1]:IF1 Best code: 1011
states[2]:IF2 Best code: 1111
states[3]:IF3 Best code: 1101
states[4]:OD Best code: 0001
states[5]:LD0 Best code: 0010
states[6]:LD1 Best code: 0011
states[7]:LD2 Best code: 0100
states[8]:ST0 Best code: 0101
states[9]:ST1 Best code: 0110
states[10]:AD0 Best code: 0111
states[11]:AD1 Best code: 1000
states[12]:AD2 Best code: 1001
states[13]:BR0 Best code: 1010
states[14]:BR1 Best code: 1100
states[15]:RES Best code: 1110

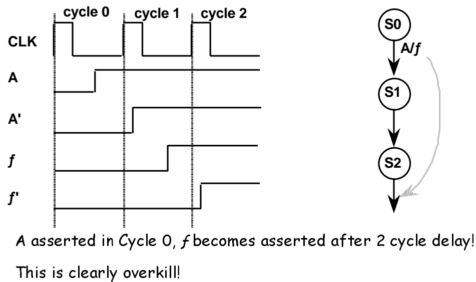
Synchronous Mealy Machines

- Standard Mealy Machine has asynchronous outputs
- These change in response to input changes, independent of clock
- Revise Mealy Machine design so outputs change only on clock edges
- One approach: non-overlapping clocks



Synchronous Mealy Machines

Case I: Synchronizers at Inputs and Outputs

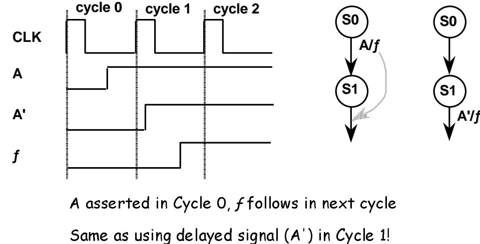


A asserted in Cycle 0, *f* becomes asserted after 2 cycle delay!

This is clearly overkill!

Synchronous Mealy Machine

Case II: Synchronizers on Inputs

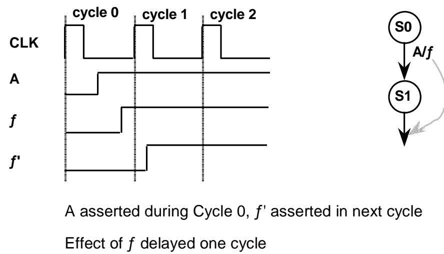


A asserted in Cycle 0, *f* follows in next cycle

Same as using delayed signal (A') in Cycle 1!

Synchronous Mealy Machines

Case III: Synchronized Outputs



A asserted during Cycle 0, *f'* asserted in next cycle

Effect of *f* delayed one cycle

Synchronous Mealy Machines

- Implications for Processor FSM Already Derived
- Consider inputs: Reset, Wait, IR<15:14>, AC<15>
  - Latter two already come from registers, and are sync'd to clock
  - Possible to load IR with new instruction in one state & perform multiway branch on opcode in next state
  - Best solution for Reset and Wait: synchronized inputs
    - » Place D flipflops between these external signals and the
    - » control inputs to the processor FSM
    - » Sync'd versions of Reset and Wait delayed by one clock cycle

Time State Divide and Conquer

- Overview
  - Classical Approach: Monolithic Implementations
  - Alternative "Divide & Conquer" Approach:
    - » Decompose FSM into several simpler communicating FSMs
    - » Time state FSM (e.g., IFetch, Decode, Execute)
    - » Instruction state FSM (e.g., LD, ST, ADD, BRN)
    - » Condition state FSM (e.g., AC < 0, AC $\neq$ 0)

Time State (Divide & Conquer)

Time State FSM

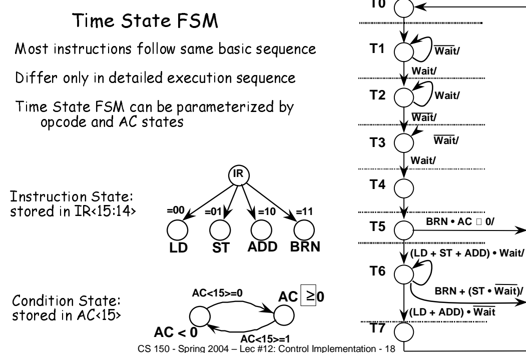
Most instructions follow same basic sequence

Differ only in detailed execution sequence

Time State FSM can be parameterized by opcode and AC states

Instruction State: stored in IR<15:14>

Condition State: stored in AC<15>



Time State (Divide & Conquer)

Generation of Microoperations

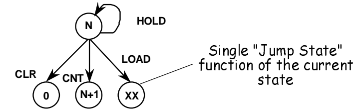
0 → PC: Reset
PC + 1 → PC: T0
PC → MAR: T0
MAR → Memory Address Bus: T2 + T6 • (LD + ST + ADD)
Memory Data Bus → MBR: T2 + T6 • (LD + ADD)
MBR → Memory Data Bus: T6 • ST
MBR → IR: T4
MBR → AC: T7 • LD
AC → MBR: T5 • ST
AC + MBR → AC: T7 • ADD
IR<13:0> → MAR: T5 • (LD + ST + ADD)
IR<13:0> → PC: T6 • BRN
1 → Read/Write: T2 + T6 • (LD + ADD)
0 → Read/Write: T6 • ST
1 → Request: T2 + T6 • (LD + ST + ADD)

Jump Counter

Concept

Implement FSM using MSI functionality: counters, mux, decoders

Pure jump counter: only one of four possible next states



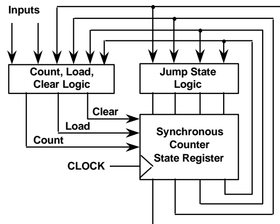
Single "Jump State" function of the current state

Hybrid jump counter:

Multiple "Jump States" — function of current state + inputs

Jump Counters

Pure Jump Counter



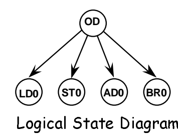
NOTE: No inputs to jump state logic

Logic blocks implemented via discrete logic, PALs/PLAs, ROMs

Jump Counters

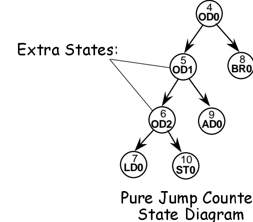
Problem with Pure Jump Counter

Difficult to implement multi-way branches



Logical State Diagram

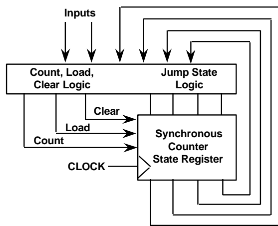
Extra States:



Pure Jump Counter State Diagram

Jump Counters

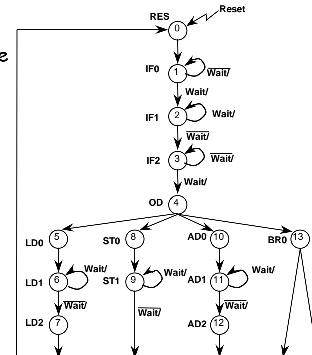
Hybrid Jump Counter



Load inputs are function of state and FSM inputs

Jump Counters

Implementation Example



State assignment attempts to take advantage of sequential states

Jump Counters

Implementation Example, Continued

CNT = (s0 + s5 + s8 + s10) + Wait • (s1 + s3) + $\overline{\text{Wait}}$ • (s2 + s6 + s9 + s11)

$\overline{\text{CNT}}$ = $\overline{\text{Wait}}$ • (s1 + s3) + Wait • (s2 + s6 + s9 + s11)

CLR = Reset + s7 + s12 + s13 + (s9 • $\overline{\text{Wait}}$)

$\overline{\text{CLR}}$ = $\overline{\text{Reset}}$ • $\overline{\text{s7}}$ • $\overline{\text{s12}}$ • $\overline{\text{s13}}$ • ($\overline{\text{s9}}$ + Wait)

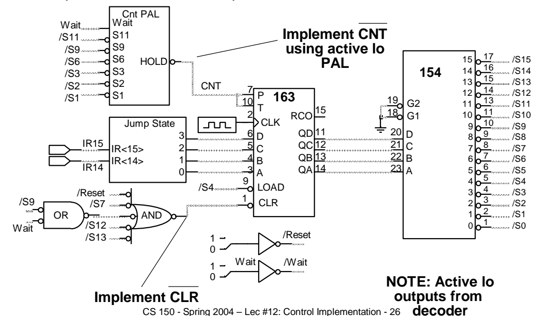
LD = s4

Contents of Jump State ROM

Address	*Contents (Symbolic State)*
00	**0101 (LD0)**
01	**1000 (ST0)**
10	**1010 (AD0)**
11	**1101 (BR0)**

Jump Counters

Implementation Example, continued

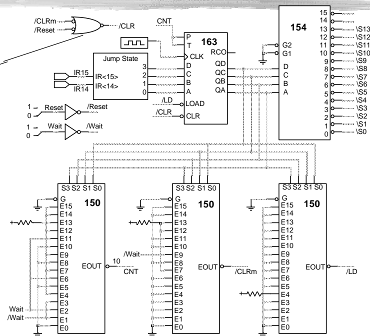


Jump Counter

CLR, CNT, LD implemented via Mux Logic

CLR = CLRm + Reset

$\overline{\text{CLR}}$ = $\overline{\text{CLRm + Reset}}$



Active Lo outputs:
hi input inverted at the output

Note that CNT is active hi on counter so invert MUX inputs!

Jump Counters

Microoperation implementation

0 → PC = Reset
PC + 1 → PC = S0
PC → MAR = S0
MAR → Memory Address Bus =
Wait•(S1 + S2 + S5 + S6 + S8 + S9 + S11 + S12)
Memory Data Bus → MBR = Wait•(S2 + S6 + S11)
MBR → Memory Data Bus = Wait•(S8 + S9)
MBR → IR = Wait•S3
MBR → AC = Wait•S7
AC → MBR = IR15•IR14•S4
AC + MBR → AC = Wait•S12
IR<13:0> → MAR = (IR15•IR14 + IR15•IR14 + IR15•IR14)•S4
IR<13:0> → PC = AC15•S13
1 → Read/Write = Wait•(S1 + S2 + S5 + S6 + S11 + S12)
0 → Read/Write = Wait•(S8 + S9)
1 → Request = Wait•(S1 + S2 + S5 + S6 + S8 + S9 + S11 + S12)

Jump Counters: CNT, CLR, LD function of current state + Wait
Why not store these as outputs of the Jump State ROM?
Make Wait and Current State part of ROM address
32 x as many words, 7 bits wide

Controller Implementation Summary

- Control Unit Organization
  - *Register transfer operation*
  - *Classical Moore and Mealy machines*
  - *Time State Approach*
  - *Jump Counter*
  - Next Time:
    - » Branch Sequencers
    - » Horizontal and Vertical Microprogramming