Overview

- Alternative controller FSM implementation approaches based on:
  - classical Moore and Mealy machines
  - jump counters
  - microprogramming (ROM) based approaches
  - branch sequencers
  - horizontal microcode
  - vertical microcode

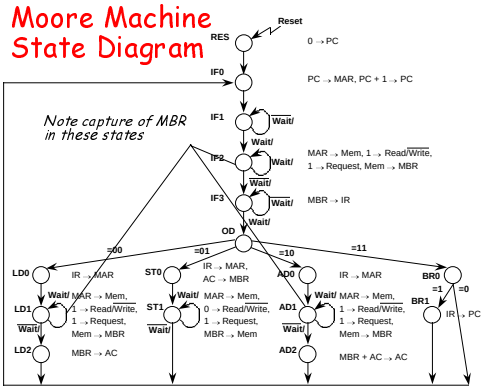
Alternative Ways to Implement Processor FSMs

- "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

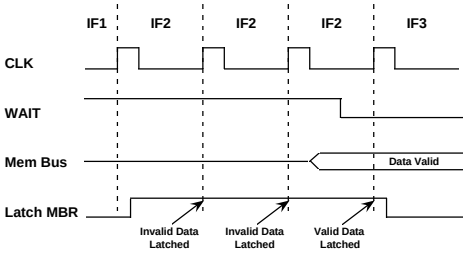
Random Logic

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

Moore Machine State Diagram

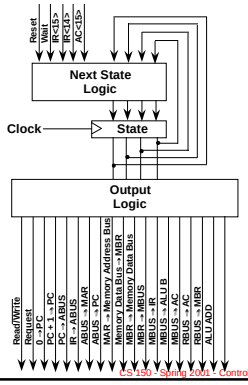


Memory-Register Interface Timing



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 → PC
0	X	X	X	X	IF0 (0001)	IF1 (0001)	PC → MAR, PC + 1 → 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 → Mem, Read,
0	0	X	X	X	IF2 (0011)	IF3 (0100)	Request, Mem → MBR
0	0	X	X	X	IF3 (0100)	IF3 (0100)	MBR → 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 → MAR
0	1	X	X	X	LD1 (0111)	LD1 (0111)	MAR → Mem, Read,
0	0	X	X	X	LD1 (0111)	LD2 (1000)	Request, Mem → MBR
0	X	X	X	X	LD2 (1000)	IF0 (0001)	MBR → AC
0	X	X	X	X	ST0 (1001)	ST1 (1010)	IR → MAR, AC → MBR
0	1	X	X	X	ST1 (1010)	ST1 (1010)	MAR → Mem, Write,
0	0	X	X	X	ST1 (1010)	IF0 (0001)	Request, MBR → Mem
0	X	X	X	X	AD0 (1011)	AD1 (1100)	IR → MAR
0	1	X	X	X	AD1 (1100)	AD1 (1100)	MAR → Mem, Read,
0	0	X	X	X	AD1 (1100)	AD2 (1101)	Request, Mem → MBR
0	X	X	X	X	AD2 (1101)	IF0 (0001)	MBR + AC → 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 → 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

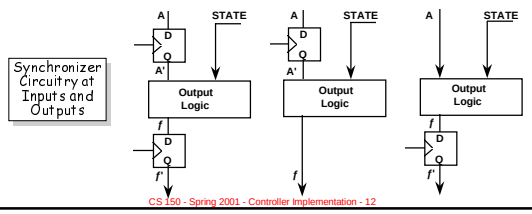
NOVA State Assignment SUMMARY

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

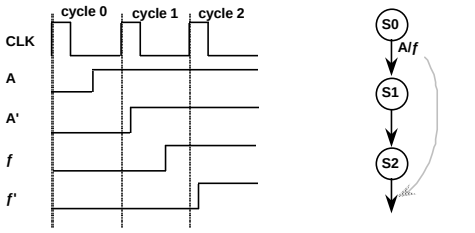
Synchronous Mealy Machines

- 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

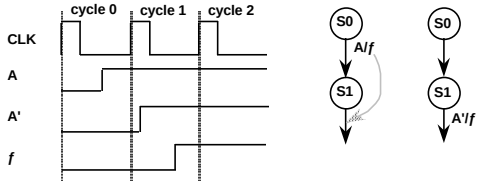


A asserted in Cycle 0, f becomes asserted after 2 cycle delay!

This is clearly overkill!

Synchronous Mealy Machine

Case II: Synchronizers on Inputs

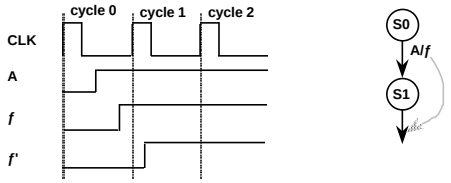


A asserted in Cycle 0, f follows in next cycle

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 ≠ 0)

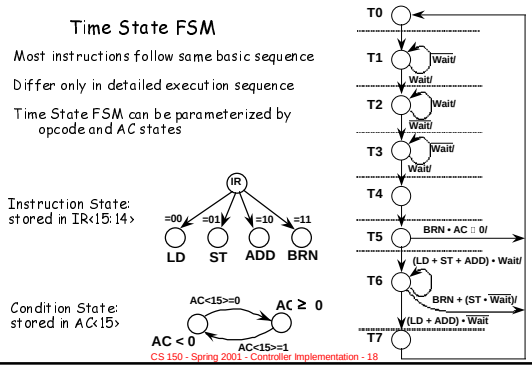
Time State (Divide & Conquer)

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



Time State (Divide & Conquer)

Generation of Microoperations

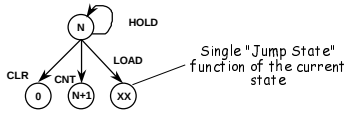
0 PC: Reset
PC + 1 PC: T0
PC MAR: T0
MAR Memory Address Bus: T2 + T6 • (LD + ST + ADD)
Memory Data Bus MBR: T2 + T6 • (LD + ADD)
MBR Memory Data Bus: T6 • ST
MBR IR: T4
MBR AC: T7 • LD
AC MBR: T5 • ST
AC + MBR AC: T7 • ADD
IR<13:0> MAR: T5 • (LD + ST + ADD)
IR<13:0> PC: T6 • BRN
1 Read/Write: T2 + T6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

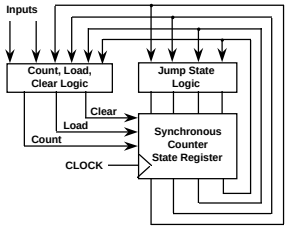
Pure jump counter: only one of four possible next states



Hybrid jump counter:
Multiple "Jump States" — function of current state + inputs

Jump Counters

Pure Jump Counter



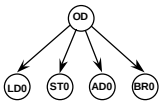
NOTE: No inputs to jump state logic

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

Jump Counters

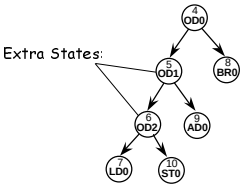
Problem with Pure Jump Counter

Difficult to implement multi-way branches



Logical State Diagram

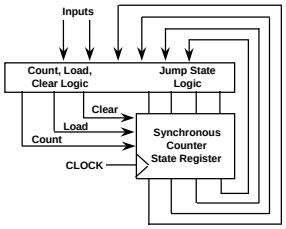
Extra States:



Pure Jump Counter State Diagram

Jump Counters

Hybrid Jump Counter

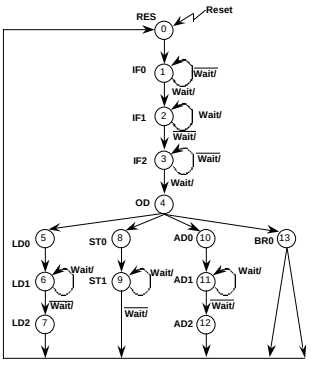


Load inputs are function of state and FSM inputs

Jump Counters

Implementation Example

State assignment attempts to take advantage of sequential states



Jump Counters

Implementation Example, Continued

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

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

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

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

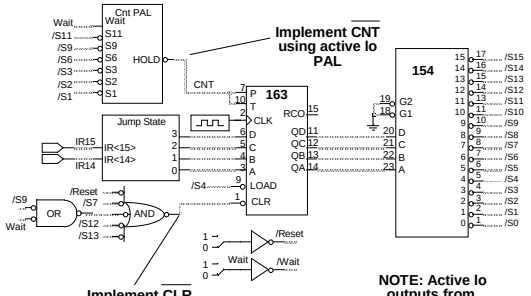
$LD = s4$

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

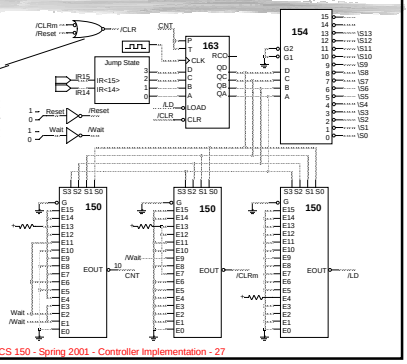
CLR, CNT, LD implemented via Mux Logic

CLR = CLRm + Reset

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

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

Branch Sequencers

Concept

Implement Next State Logic via ROM

Address ROM with current state and inputs

Problem: ROM doubles in size for each additional input

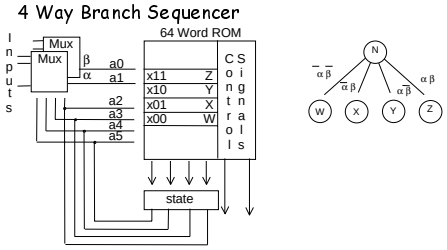
Note: Jump counter trades off ROM size vs. external logic
- Only jump states kept in ROM
- Even in hybrid approach, state + input subset form ROM address

Branch Sequencer: between the extremes
- Next State stored in ROM
- Each state limited to small number of next states
- Always a power of 2

Observe: only a small set of inputs are examined in any state

Branch Sequencers

4 Way Branch Sequencer



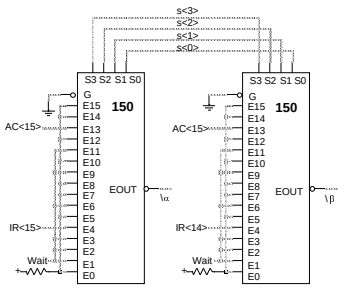
Current State selects two inputs to form part of ROM address

These select one of four possible next states (and output sets)

Every state has exactly four possible next states

Branch Sequencer

Processor CPU Design Example



Alpha, Beta multiplexer input setup

Example Processor FSM

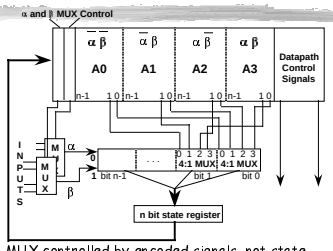
ROM ADDRESS (Reset, Current State, a, b)					ROM CONTENTS Next State	Register Transfer Operations		
RES	0	0000	X	X	0001 (IF0)	PC	MAR, PC + 1	PC
IF0	0	0001	0	0	0001 (IF0)			
	0	0001	1	1	0010 (IF1)	MAR	Mem, Read, Request	
IF1	0	0010	0	0	0011 (IF2)	MAR	Mem, Read, Request	
	0	0010	1	1	0010 (IF1)	Mem	MBR	
IF2	0	0011	0	0	0011 (IF2)			
	0	0011	1	1	0100 (OD)	MBR	IR	
OD	0	0100	0	0	0101 (LD0)	IR	MAR	
	0	0100	0	1	1000 (ST0)	IR	MAR, AC	MBR
	0	0100	1	0	1001 (AD0)	IR	MAR	
	0	0100	1	1	1101 (BR0)	IR	MAR	

Example Processor FSM

ROM ADDRESS (Reset, Current State, a, b)					ROM CONTENTS Next State	Register Transfer Operations			
LD0	0	0101	X	X	0110 (LD1)	MAR	Mem, Read, Request		
LD1	0	0110	0	0	0111 (LD2)	Mem	MBR		
	0	0110	1	1	0110 (LD1)	MAR	Mem, Read, Request		
LD2	0	0111	X	X	0000 (RES)	MBR	AC		
ST0	0	1000	X	X	1001 (ST1)	MAR	Mem, Write, Request, MBR		Mem
ST1	0	1001	0	0	0000 (RES)				
	0	1001	1	1	1001 (ST1)	MAR	Mem, Write, Request, MBR		Mem
AD0	0	1010	X	X	1011 (AD1)	MAR	Mem, Read, Request		
AD1	0	1011	0	0	1100 (AD2)				
	0	1011	1	1	1011 (AD1)	MAR	Mem, Read, Request		
AD2	0	1100	X	X	0000 (RES)	MBR + AC	AC		
BR0	0	1101	0	0	0000 (RES)				
	0	1101	1	1	0000 (RES)	IR	PC		

Branch Sequencers

Alternative Horizontal Implementation



Input MUX controlled by encoded signals, not state
Much fewer inputs than unique states!
In example FSM, input MUX can be 2:1!

Adding length to ROM word saves on bits vs. doubling words
Vertical format: (14 + 4) x 64 = 1152 ROM bits
Horizontal format: (14 + 4 x 4 + 2) x 16 = 512 ROM bits

Microprogramming

How to organize the control signals

Implement control signals by storing 1's and 0's in a ROM

Horizontal vs. vertical microprogramming

Horizontal: 1 ROM output for each control signal

Vertical: encoded control signals in ROM, decoded externally
some mutually exclusive signals can be combined
helps reduce ROM length

Microprogramming

Register Transfer/Microoperations

14 Register Transfer operations become 22 Microoperations:

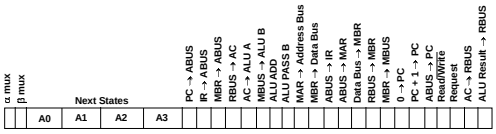
PC ABUS
IR ABUS
MBR ABUS
RBUS AC
AC ALU A
MBUS ALU B
ALU ADD
ALU PASS B
MAR Address Bus
MBR Data Bus
ABUS IR
ABUS MAR
Data Bus MBR
RBUS MBR
MBR MBUS
0 PC
PC + 1 PC
ABUS PC
Read/Write
Request
AC RBUS
ALU Result RBUS

Horizontal Microprogramming

Horizontal Branch Sequencer

- Mux bits
- 4 x 4 Next State bits
- 22 Control operation bits

40 bits total



Horizontal Microprogramming

Moore Processor ROM

Current State (Address)	α mux	β mux	Next States A0	A1	A2	A3	PC → ABUS	IR → ABUS	MBR → ABUS	RBUS → AC	AC → ALU A	MBUS → ALU B	ALU ADD	ALU PASS B	MAR → Address Bus	MBR → Data Bus	ABUS → IR	ABUS → MAR	Data Bus → MBR	RBUS → MBR	MBR → MBUS	0 → PC	PC + 1 → PC	ABUS → PC	Read/Write	Request	AC → RBUS	ALU Result → RBUS
RES (0000)	0	0	0001	0001	0001	0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
IF0 (0001)	0	0	0010	0010	0010	0010	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
IF1 (0010)	0	0	0010	0010	0011	0011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
IF2 (0011)	0	0	0100	0100	0011	0011	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0
IF3 (0100)	0	0	0100	0100	0101	0101	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
OD (0101)	1	1	0110	1001	1011	1110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LD0 (0110)	0	0	0111	0111	0111	0111	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
LD1 (0111)	0	0	1000	1000	0111	0111	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0	0
LD2 (1000)	0	0	0001	0001	0001	0001	0	0	0	1	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1
ST0 (1001)	0	0	1010	1010	1010	1010	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0
ST1 (1010)	0	0	0001	0001	1010	1010	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0
AD0 (1011)	0	0	1100	1100	1100	1100	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
AD1 (1100)	0	0	1101	1101	1100	1100	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0	0
AD2 (1101)	0	0	0001	0001	0001	0001	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
BR0 (1110)	0	1	0001	1111	0001	1111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
BR1 (1111)	0	0	0001	0001	0001	0001	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Alpha inputs: 0 = Wait, 1 = IR<15>
Beta inputs: 0 = AC<15>, 1 = IR<14>

Horizontal Microprogramming

Advantages:
most flexibility -- complete parallel access to datapath control points

Disadvantages:
very long control words -- 100+ bits for real processors

NOTE: Not all microoperation combinations make sense!

Output Encodings:
Group mutually exclusive signals
Use external logic to decode

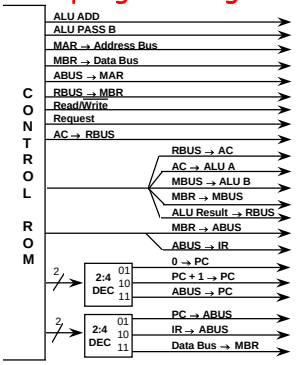
Example:

0 PC, PC + 1 PC, ABUS PC mutually exclusive

Save ROM bit with external 2:4 Decoder

Horizontal Microprogramming

Partially Encoded Control Outputs



Vertical Microprogramming

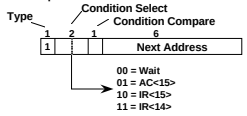
More extensive encoding to reduce ROM word length

- Typically use multiple microword formats:
  - Horizontal microcode -- next state + control bits in same word
  - Separate formats for control outputs and "branch jumps"
  - may require several microwords in a sequence to implement same function as single horizontal word
- In the extreme, very much like assembly language programming

Vertical Microprogramming

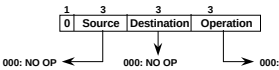
Branch Jump
Compare indicated signal to 0 or 1

Branch Jump Format



Register Transfer
Source, Destination, Operation

Register Transfer Format



000: NO OP
001: PC → ABUS
010: IR → ABUS
011: MBR → MBUS
100: MAR → M
101: AC → RBUS
110: ALU Res → RBUS

000: NO OP
001: RBUS → AC
010: MBUS → IR
011: ABUS → MAR
100: M → MBR
101: RBUS → MBR
110: ABUS → PC
111: MBR → M

000: NO OP
001: ALU ADD
010: ALU PASS B
011: 0 → PC
100: PC + 1 → PC
101: Read
110: Write

10 ROM Bits

Vertical Microprogramming

ROM ADDRESS	SYMBOLIC CONTENTS	BINARY CONTENTS
000000	RES RT PC MAR, PC+1 PC	0 001 011 100
000001	IF0 RT MAR M, Read	0 100 000 101
000010	BJ Wait=0, IF0	1 000 000 001
000011	IF1 RT MAR M, M MBR, Read	0 100 100 101
000100	BJ Wait=1, IF1	1 001 000 011
000101	IF2 RT MBR IR	0 011 010 000
000110	BJ Wait=0, IF2	1 000 000 101
000111	RT IR MAR	0 010 011 000
001000	OD BJ IR<15>=1, OD1	1 101 010 101
001001	BJ IR<14>=1, ST0	1 111 010 000
001010	LD0 RT MAR M, Read	0 100 000 101
001011	LD1 RT MAR M, M MBR, Read	0 100 100 101
001100	BJ Wait=1, LD1	1 001 001 011
001101	LD2 RT MBR AC	0 110 001 010
001110	BJ Wait=0, RES	1 000 000 000
001111	BJ Wait=1, RES	1 001 000 000

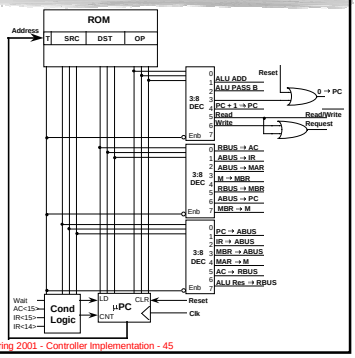
Vertical Microprogramming

ROM ADDRESS	SYMBOLIC CONTENTS	BINARY CONTENTS
010000	ST0 RT AC MBR	0 101 101 000
010001	RT MAR M, MBR M, Write	0 100 111 110
010010	ST1 RT MAR M, MBR M, Write	0 100 111 110
010011	BJ Wait=0, RES	1 000 000 000
010100	BJ Wait=1, ST1	1 001 010 010
010101	OD1 BJ IR<14>=1, BR0	1 111 011 101
010110	AD0 RT MAR M, Read	0 100 000 101
010111	AD1 RT MAR M, M MBR, Read	0 100 100 101
011000	BJ Wait=1, AD1	1 001 010 111
011001	AD2 RT AC+MBR AC	0 110 001 001
011010	BJ Wait=0, RES	1 000 000 000
011011	BJ Wait=1, RES	1 000 000 000
011100	BR0 BJ AC<15>=0, RES	1 010 000 000
011101	RT IR PC	0 010 110 000
011110	BJ AC<15>=1, RES	1 011 000 000

31 words x 10 ROM bits = 310 bits total *versus* 16 x 38 = 608 bits horizontal

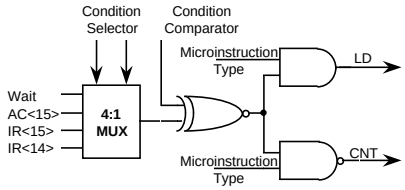
Vertical Programming

Controller Block Diagram



Vertical Microprogramming

Condition Logic



Vertical Microprogramming

- Writeable Control Store
  - Part of control store addresses map into RAM
    - » Allows assembly language programmer to implement own instructions
    - » Extend "native" instruction set with application specific instructions
    - » Requires considerable sophistication to write microcode
    - » Not a popular approach with today's processors
  - Make the native instruction set simple and fast
  - Write "higher level" functions as assembly language sequences

Controller Implementation Summary

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