Computer and VLSI design

Control, finite state machines, microprogramming

P.J. Drongowski
SandSoftwareSound.net

Copyright © 1987-2013 Paul J. Drongowski

Control implementation styles

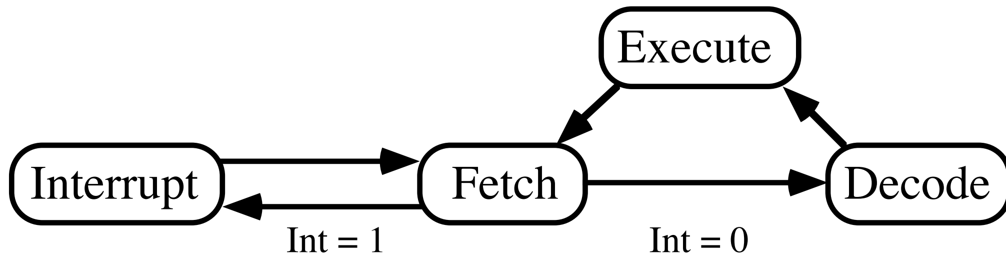
- Finite state machine (FSM)
    - "Random logic" or "hardwired" approach
    - Draw state diagram for ISA instruction interpretation
    - Perform state assignment
    - Define state table
    - Implement sequential logic circuit
    - Implementation is usually ill-structured and not regular
    - PLA can make design more regular and structured

- Microprogramming
    - Each control step can be represented as an instruction
    - These control instructions can be stored in a memory
    - Technique developed by Maurice Wilkes in 1951
    - Terminology
        - ◊ Microprogram
        - ◊ Microinstruction
        - ◊ Emulation (interpeter for the ISA)
    - Control store
        - ◊ Microprogram memory
        - ◊ May be writeable (not just read-only)
        - ◊ Update to correct errors

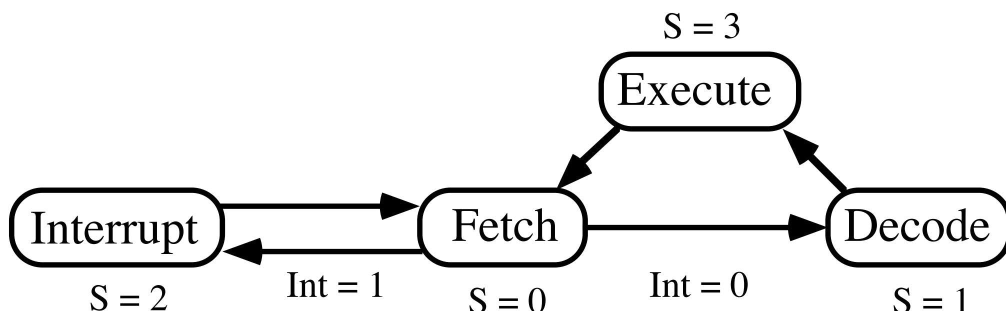
- Two styles of microprogramming
    - Vertical
    - Horizontal

Random logic controller

- Consider the state diagram below



- Four state machine
  - Fetch state - get instruction from memory
  - Decode state - decode new instruction
  - Execute state - execute decoded instruction
  - Interrupt state - perform trap sequence
- Signal "Int" is an external input
  - Signal is asserted when an interrupt is requested
  - Go to Interrupt state if asserted
  - Otherwise, go to Decode state
- First step in controller design is state assignment
  - Fetch ⇒ 0
  - Decode ⇒ 1
  - Interrupt ⇒ 2
  - Execute ⇒ 3
- Redraw the state diagram with states assigned
- Assume two flip / flop (bit) representation of state

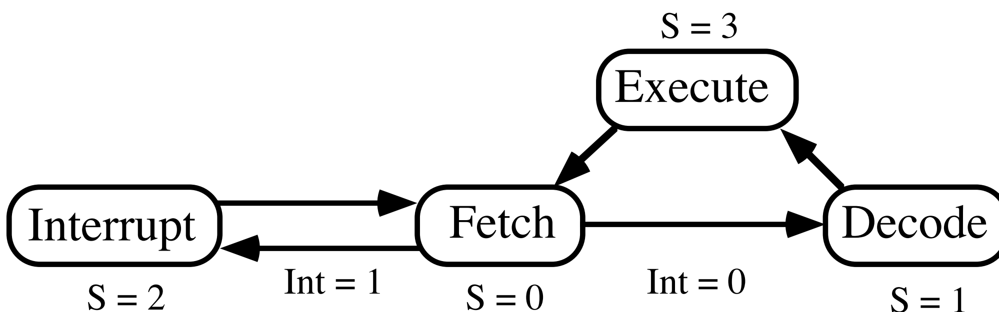


State table form

- After state assignment, build equivalent state table
- Each row in table represents:
  - Current machine state, and
  - Condition input combination

State	Int	S1	S0	
0	0	0	0	
0	1	0	0	
1	X	0	1	
2	X	1	0	
3	X	1	1	

- S1 and S1 represent high and low bit of the machine state
- "X" terms denote "don't care" condition
- Don't care terms specify input conditions to be ignored
- Each row will become a product term
  - $\sim Int \wedge \sim S1 \wedge \sim S0$ (First row term)
  - $Int \wedge \sim S1 \wedge \sim S0$ (Second row term)
  - $\sim S1 \wedge S0$ (Third row term)
  - $S1 \wedge \sim S0$ (Fourth row term)
  - $S1 \wedge S0$ (Fifth row term)
- Note that ignored inputs don't appear in product term
- The product term will enable the rest of the table

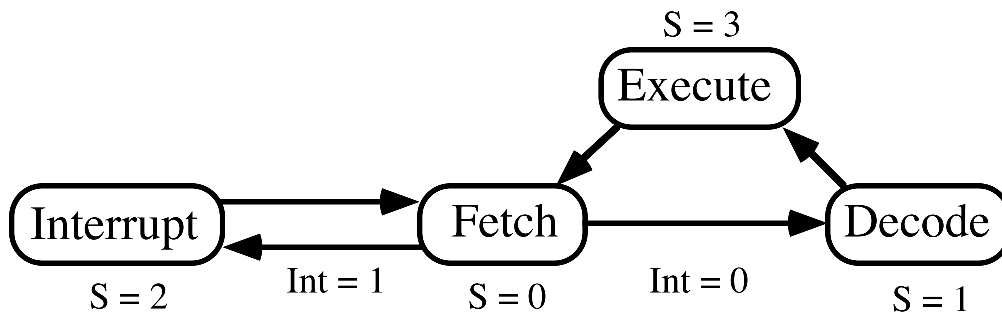


Filling out the table

- For a given state and input:
  - Select a next state value
  - Generate control signals to datapath

State	Int	S1	S0	Next1	Next0	Control
0	0	0	0	0	1	1010101
0	1	0	0	1	0	0001100
1	X	0	1	1	1	0100101
2	X	1	0	0	0	0111110
3	X	1	1	0	0	1110001

- Next state values are feed back through state flip / flops
- Next1, Next0, Control ouputs are formed by OR logic



- Inputs to the table are variables in product terms (AND)
- Outputs are formed by OR'ing across columns
- Thus, each output signal is a sum of products expression
- Programmable logic array (PLA)
- The table can be implemented directly in a PLA

Finite state machines

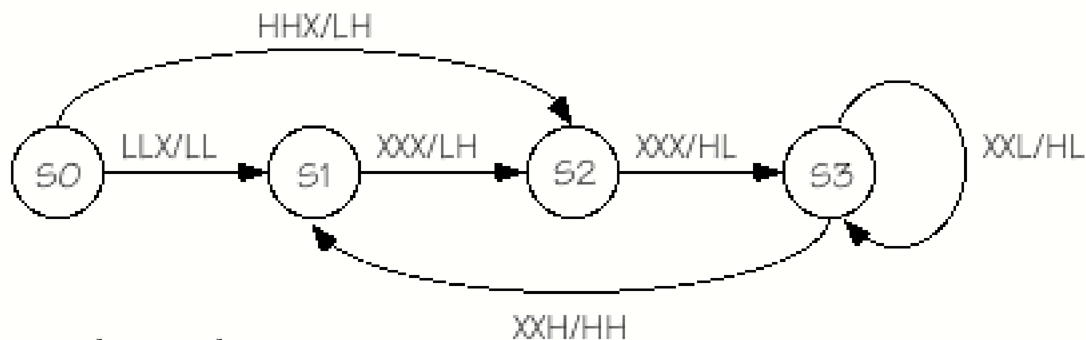
- Common class of computations
- State table is one possible representation
- Each row in table contains
  - Current inputs
  - Current machine state
  - Next machine state
  - New outputs

Inputs A B C	This state	Next state	Outputs X Y
LLX	S0	S1	LL
HHX	S0	S2	LH
XXX	S1	S2	LH
XXX	S2	S3	HL
XXL	S3	S3	HL
XXH	S3	S1	HH

- Operation
  - Find row matching current state and inputs
  - Treat X as a "*don't care*" - ignore that input
  - Assert the specified output values
  - Go to the next state making it the new current state
  - Iterate

Alternate representations

- State diagram
  - States are represented by labelled circles
  - An arc shows transition from one state to another
  - Arcs are labelled with inputs / outputs



- Pseudo-code

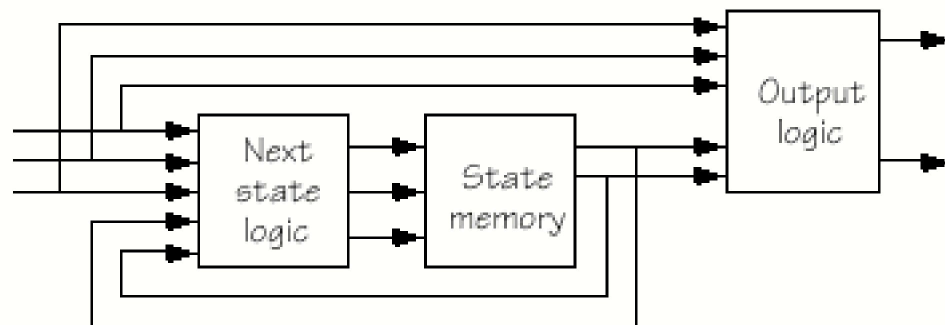
```
input ABC
output XY
states S0, S1, S2, S3

S0: if ABC = LLX then XY := LL ; goto S1
    if ABC = HHX then XY := LH ; goto S2
S1: XY := LH ; goto S2
S2: XY := HL ; goto S3
S3: if ABC = XXL then XY := HL ; goto S3
    if ABC = XXH then XY := HH ; goto S1
```

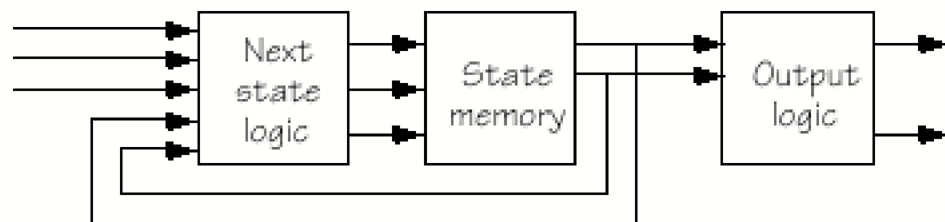
- Are all input states covered? S0: LHX or HLX?
- Horizontal microcode
  - Each microinstruction is a "row" in the state table
  - State table is held in ROM, RAM or PLA
  - Input conditions encoded in addressing or selection
  - Often uses next μaddress field (next state)
- Lexical analysis / parse table
  - Input is next character or symbol in scan
  - States / transitions to recognize composite constructs like numbers, identifiers, etc.
  - Instead of outputs, perform actions like store character, accumulate digit value, etc.

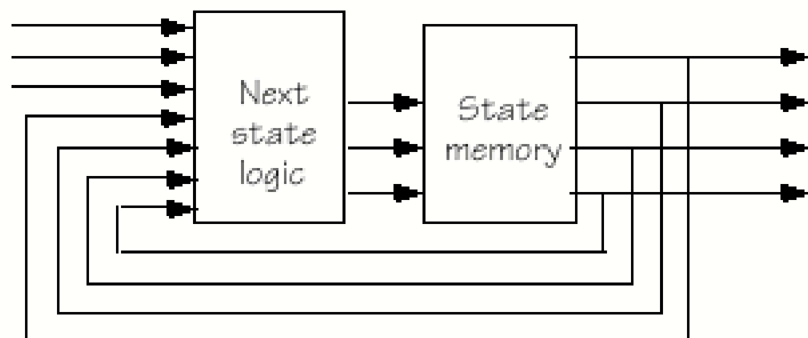
FSM types

- Mealy machine
  - Named after G.H. Mealy
  - Outputs are function of present input and state



- Moore machine
  - Named after E.F. Moore
  - Output is strictly a function of machine state

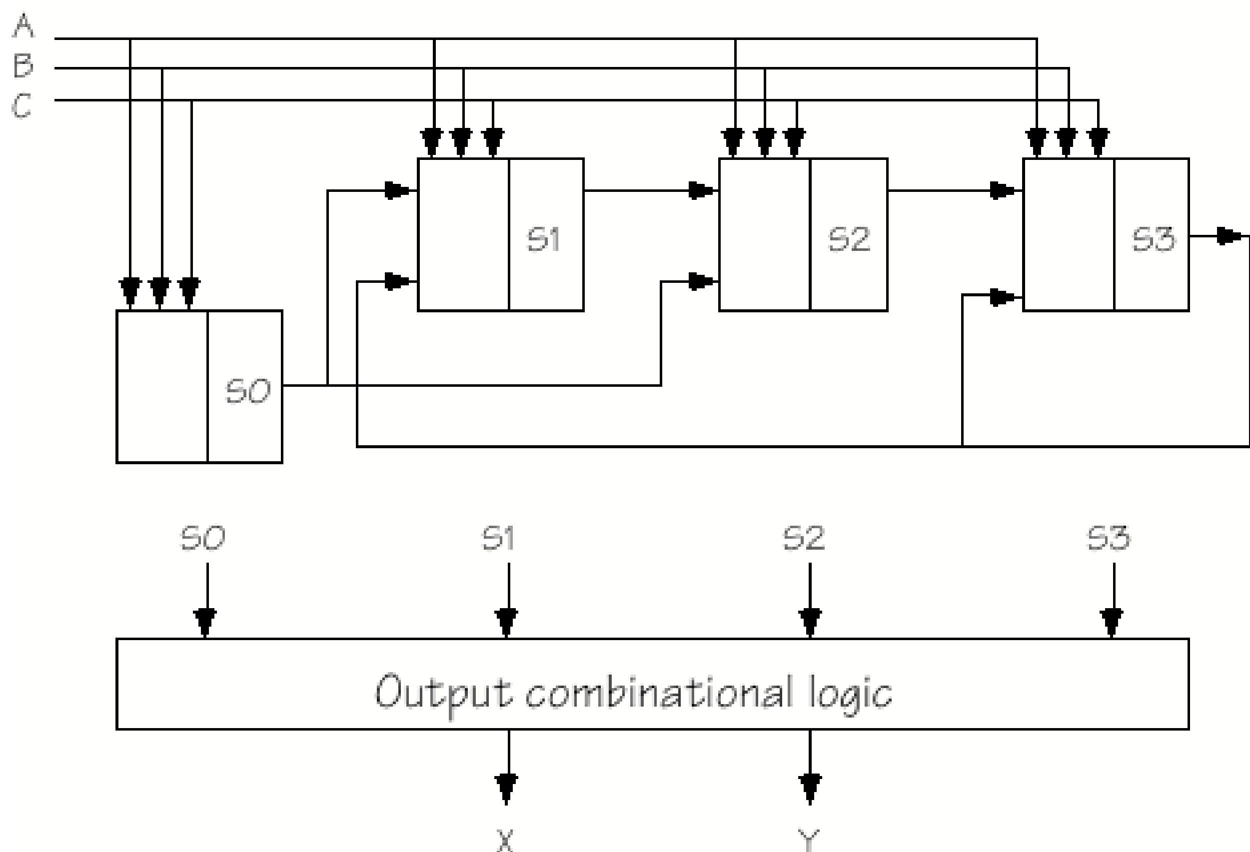




Flip / flop per state

- Assign a flip / flop to each state
- Flip / flop is one when state is active (current)
- Sometimes called "one hot" sequential logic
- Need logic between state flip / flops

Inputs A B C	This state	Next state	Outputs X Y
LLX	S0	S1	LL
HHX	S0	S2	LH
XXX	S1	S2	LH
XXX	S2	S3	HL
XXL	S3	S3	HL
XXH	S3	S1	HH



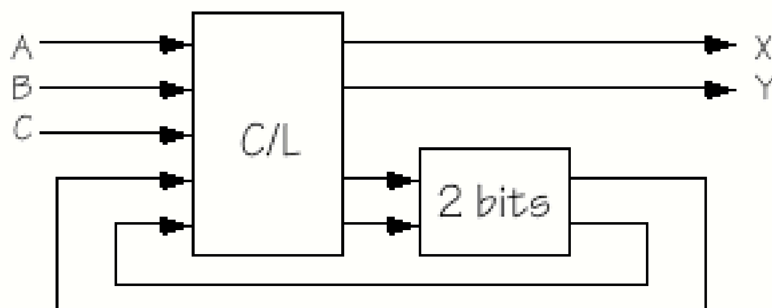
Encoded state

- Use an N-bit register to hold current state
- Assign each state a unique binary number
- Assume master - slave F/F operation

Inputs A B C	This state	Next state	Outputs X Y
LLX	S0	S1	LL
HHX	S0	S2	LH
XXX	S1	S2	LH
XXX	S2	S3	HL
XXL	S3	S3	HL
XXH	S3	S1	HH

Assign states

S0 → 0
S1 → 1
S2 → 2
S3 → 3



- Two master - slave flip / flops
  - Capture new state on rising edge
  - Change out to new state value on trailing edge
- Design problem centers on combinational logic
  - Five inputs: A, B, C, Q1, and Q0
  - Four outputs: X, Y, D1, and D0
  - Four sum of product equations (one per output)
  - Equations and logic could be minimized

Encoded state (2)

- Re-write state table with D1, DO, Q1 and QO

Inputs A B C	This D1 DO	Next Q1 QO	Outputs X Y
LLX	LL	LH	LL
HHX	LL	HL	LH
XXX	LH	HL	LH
XXX	HL	HH	HL
XXL	HH	HH	HL
XXH	HH	LH	HH

- Write sum of products equations Q1, QO, X and Y

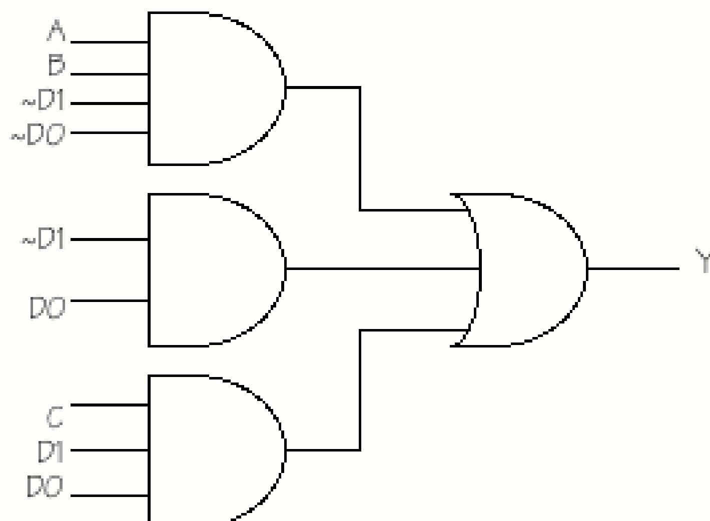
Q1 = A•B•~D1•~DO + ~D1•DO + D1•~DO + ~C•D1•DO

QO = ~A•~B•~D1•~DO + D1•~DO + ~C•D1•DO + C•D1•DO

X = D1•~DO + ~C•D1•DO + C•D1•DO = D1

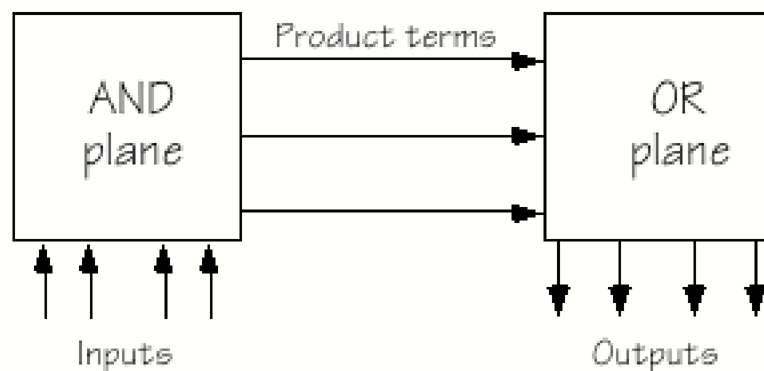
Y = A•B•~D1•~DO + ~D1•DO + C•D1•DO

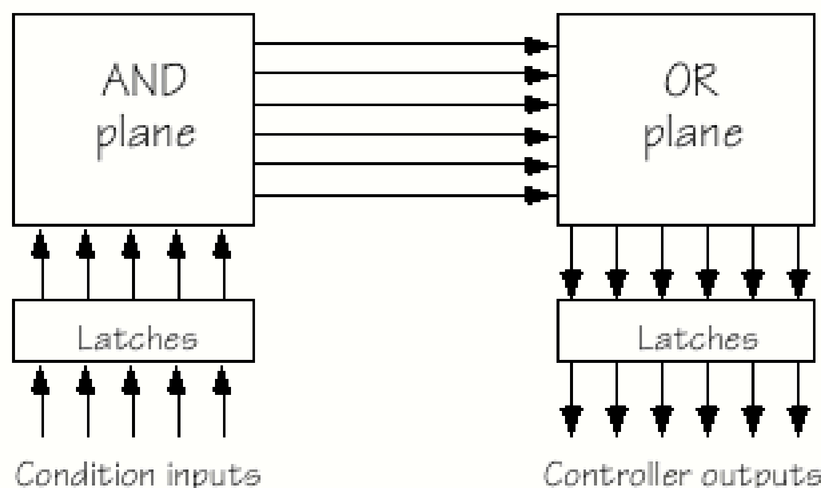
- Reduce and build gate network for each equation



Programmable logic arrays

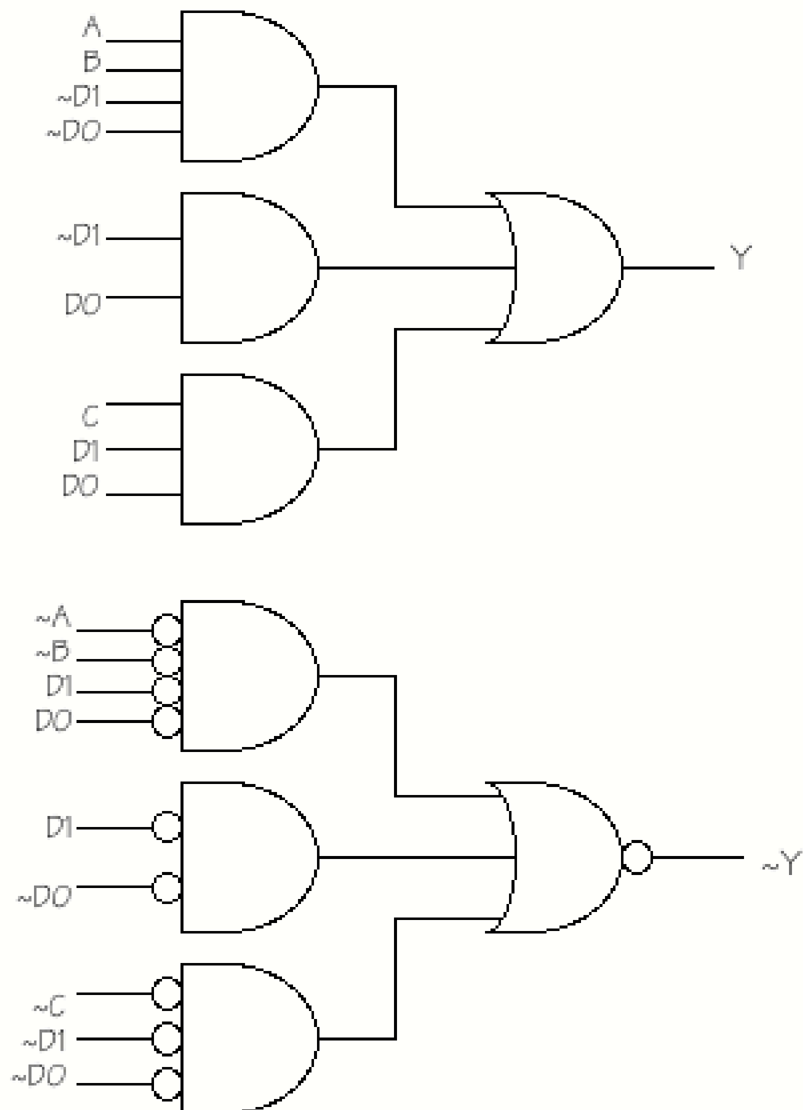
- Often need to implement block of combinational logic
- Sum of products is really a two-level gate network
- Programmable logic array (PLA)
  - Two-level gate network
  - Provides a general framework for AND/OR logic
  - Separated into AND- and OR-planes
  - Framework forms true/complement of inputs
  - Program by adding transistors or connections
  - Add input and output latches to form FSM
- Advantages
  - Programmability, generation tools (CAD)
  - Regular, compact layout





NOR-NOR PLA

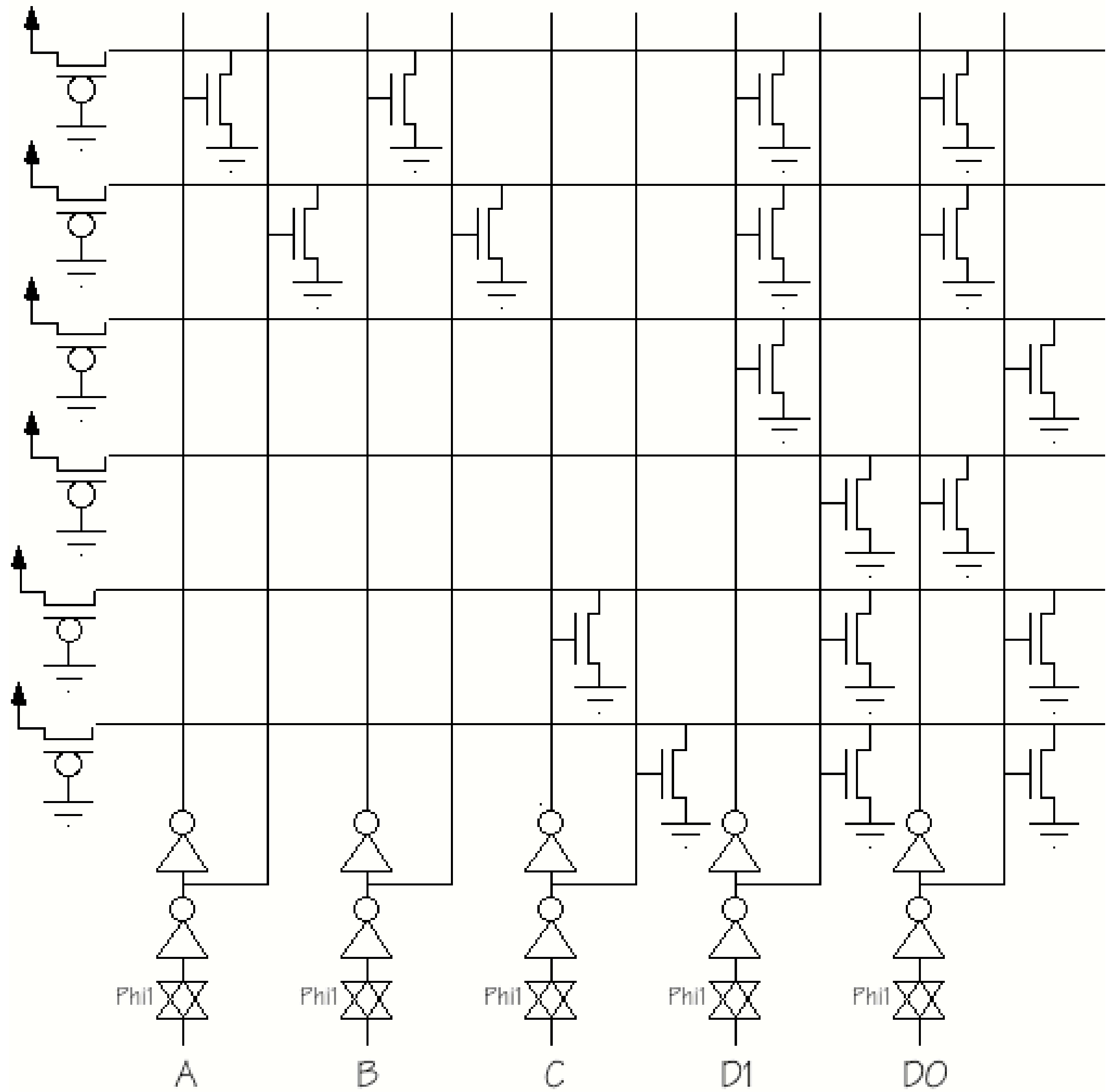
- Typically use NOR-NOR logic
  - NAND array is slower
  - Long series chain of switches should be avoided
- Need to invert input terms (de Morgan's theorem)
- Not expensive as PLA's usually generate both true and complement forms of the inputs anyway



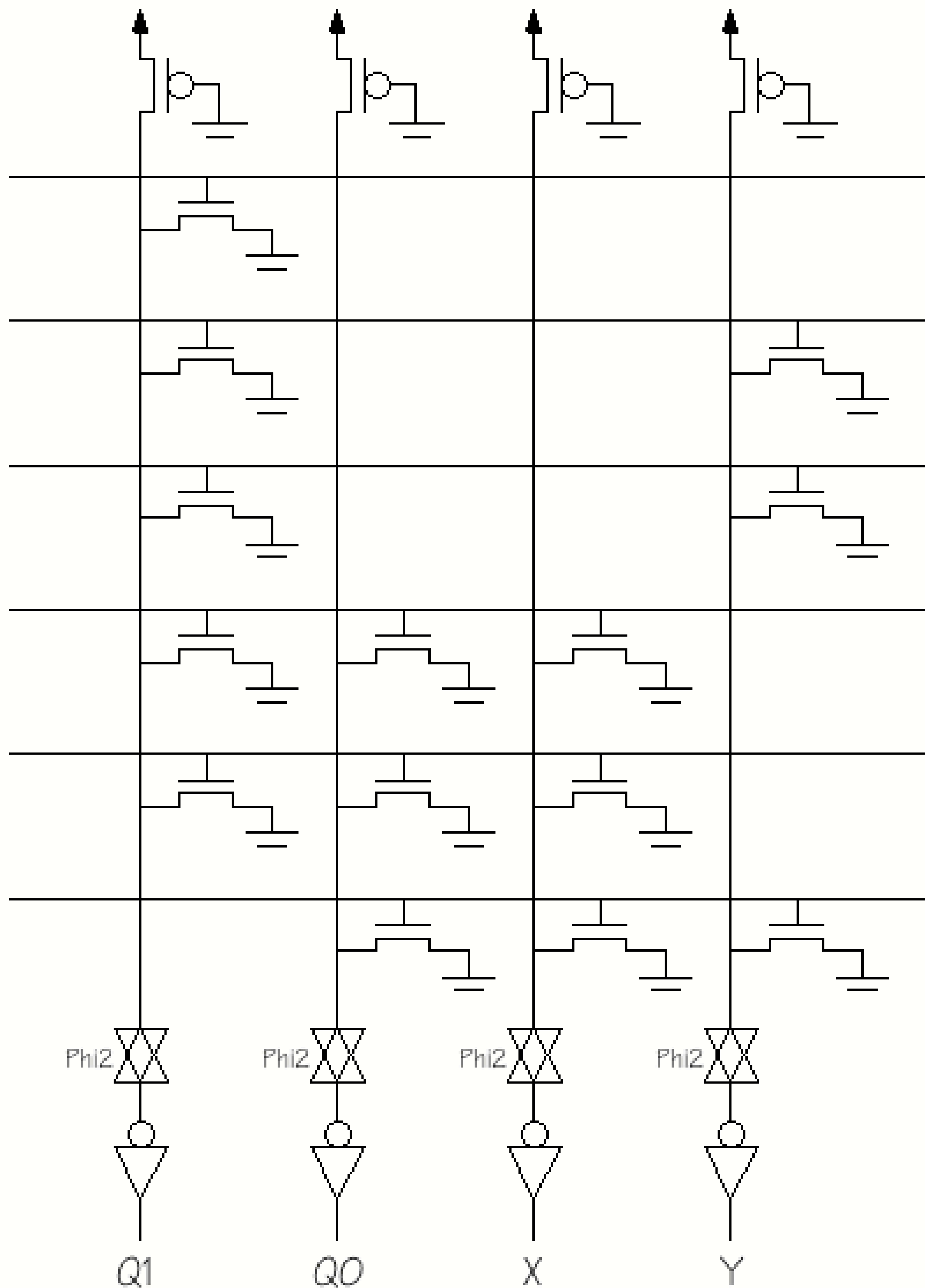
CMOS PLA implementations

- CMOS VLSI Design, Weste & Eshraghian, pg. 368
- Complementary gates complicate PLA layout
- Corresponding n- and p-gates must be connected
- Pseudo-nMOS
  - nMOS layout is simpler
  - Complementary gates not connected
  - Use p-channel device as pull-up resistor
  - Tie p-channel gate to ground; channel always ON
  - Requires ratio'd logic; choose L and W carefully
  - Static power consumption much higher
- Dynamic CMOS - 2-phase clocking
  - Latch inputs on Phi-1
  - Latch outputs on Phi-2
  - Precharge product term lines during ~Phi-1
  - Set product term lines during Phi-1 (pull-down)
  - Approach #1
    - Generate internal OR-plane precharge clock
    - Must accommodate worst case loading delay
    - Self-timed logic
    - Precharge OR-plane when internal clock is low
    - Set output lines when internal clock asserted
  - Approach #2
    - Insert latch between AND and OR planes
    - Latch product terms on Phi-2
    - Latch OR-plane outputs on Phi-1
    - Pipeline register puts PLA one cycle behind
- Four-phase clocking
  - Extra intermediate latches
  - Must carefully apply timing discipline to design

Example AND-plane



Example OR-plane

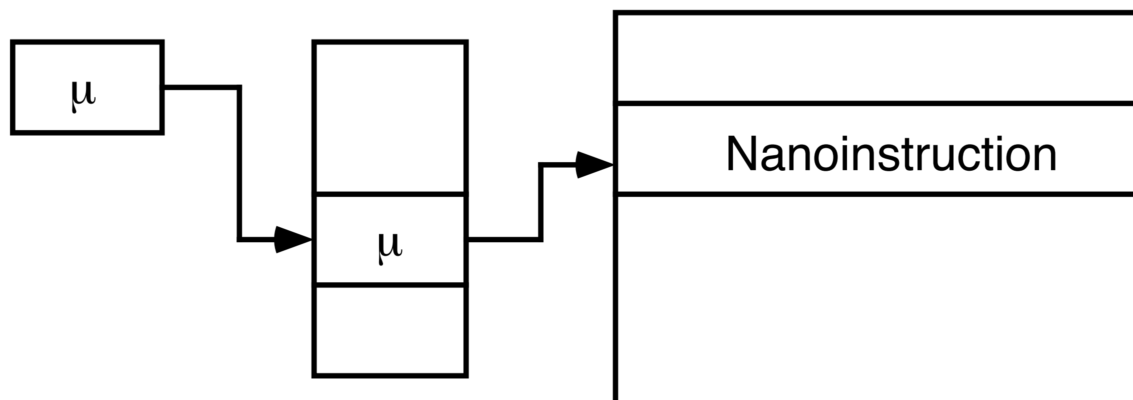


Microprogramming styles

- Horizontal microprogramming
  - Low degree of encoding
  - Apply microinstruction bits directly to control inputs
  - No encoding ⇒ less control delay
  - Results in very long microwords (32 to 300 bits!)
  - Exploitable parallelism
  - Theory to optimize microprogram size

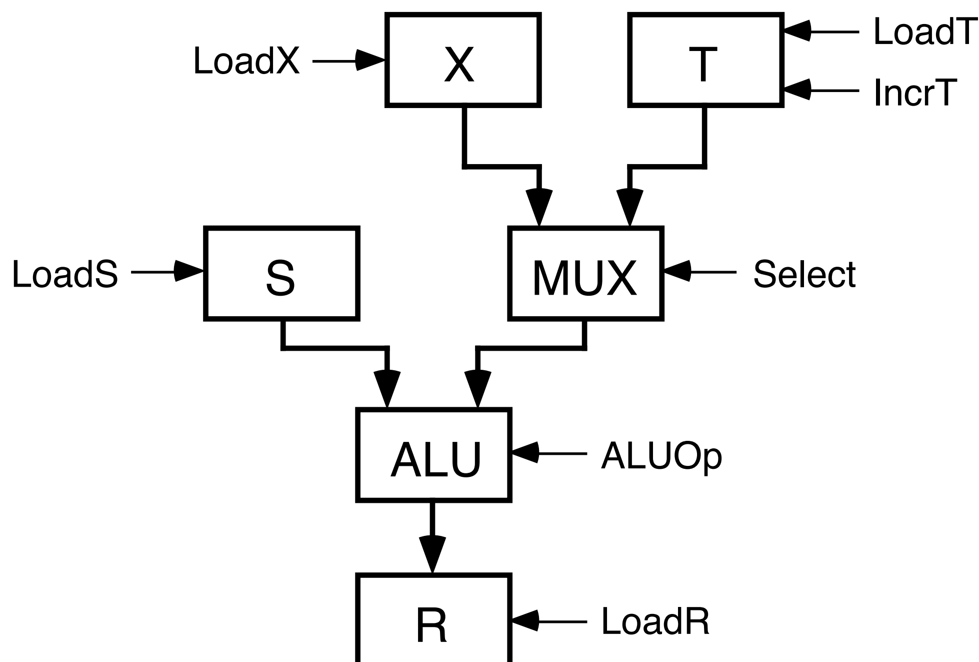
- Vertical microprogramming
  - High degree of instruction encoding
  - Instructions resemble ordinary (ISA) programming
  - Microwords are short (16 to 32 bits)
  - Short microword ⇒ smaller control store
  - Typically cannot exploit as much parallelism

- Two-level control store (hybrid)
  - Take advantage of redundancy in control store
  - Short microinstruction indexes table of *nano* instructions
  - Nanoinstructions are unique *control states*
  - Apply nanoinstructions directly to control inputs
  - Example: Motorola 68000
    - ◊ Single level store: 52,400 bits
    - ◊ Two level store (total): 30,550 bits



Microinstruction design

- Make a list of all control signals in the datapath
    - ◊ Signal name(s)
    - ◊ Operation ⇔ signal value table
- Add sequencing control signals
    - ◊ Next μ instruction address (branch address)
    - ◊ Branch condition selector
- Draw μ word format
    - ◊ Assign signals to μ instruction fields



LoadX
    0: No operation
    1: X ← new value
LoadS
    0: No operation
    1: S ← new value
LoadT
    0: No operation
    1: T ← new value
IncrT
    0: No operation

LoadR
    0: No operation
    1: R ← ALU result
ALUOp
    0: No operation
    1: Add
    2: Subtract
    3: AND
    4: etc.
Select
    0: Pass X
    1: Pass T

Microinstruction encoding

- Encode each graph block into one μ instruction
- Find datapath elements participating into transfers
- Select control signal values for each transfer

LoadX
- 0: No operation
- 1: X $\leftarrow$ new value

LoadS
- 0: No operation
- 1: S $\leftarrow$ new value

LoadT
- 0: No operation
- 1: T $\leftarrow$ new value

IncrT
- 0: No operation

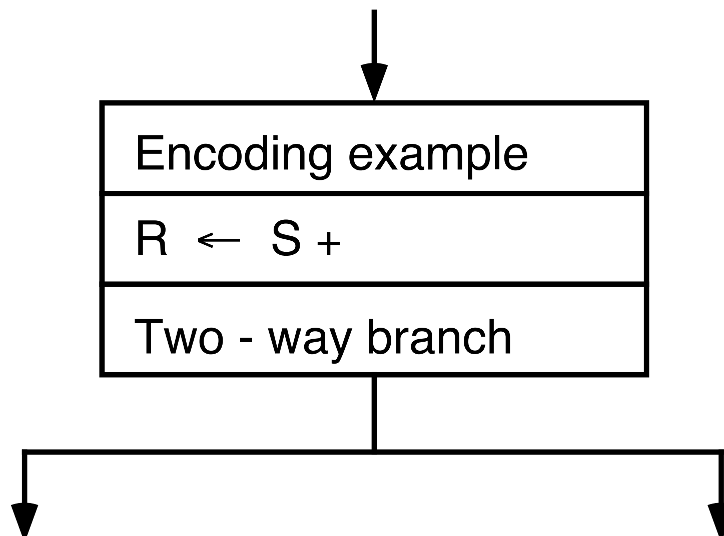
LoadR
- 0: No operation
- 1: R $\leftarrow$ ALU result

ALUOp
- 0: No operation
- 1: Add
- 2: Subtract
- 3: AND
- 4: etc.

Select
- 0: Pass X
- 1: Pass T

LoadX	LoadS	LoadT	IncrT	LoadR
0	0	0	0	1

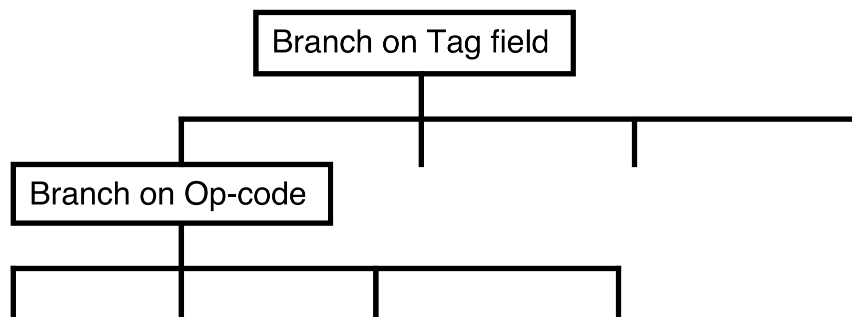
ALUOp		Select	Branch		
0	1	1	0	1	0



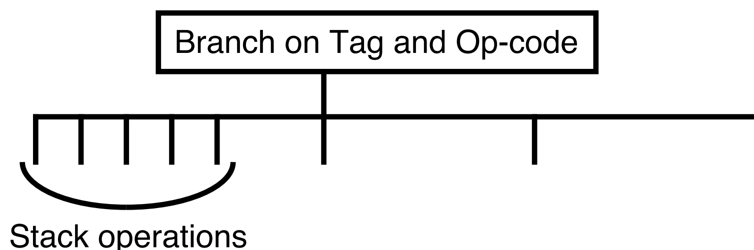
Levels of decoding

- ISA with multiple operation fields
- SP.3 example
  - Instruction tag field {stack, literal, op and desc call}
  - Multiple stack operations

- Two levels of decode



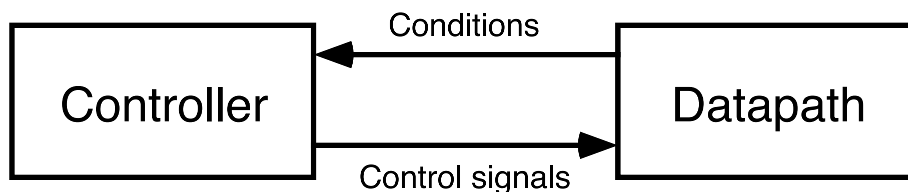
- One level of decode



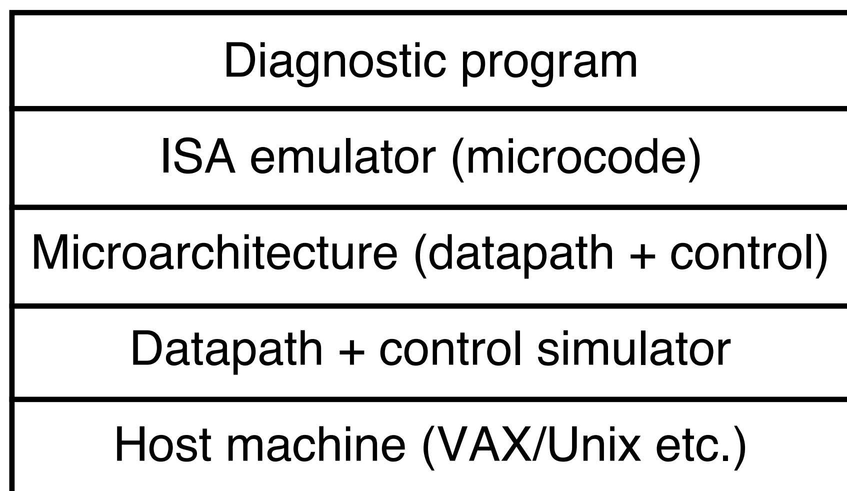
- Move instruction
  - Most frequently executed instruction
  - Make instruction as fast as possible
  - Two level approach
    - Decode and identify move operation
    - Decode and identify addressing mode
    - Execute
  - One level approach
    - Decode and identify move absolute
    - Execute
  - Considerable savings will accrue

Overall design style

- Central, synchronous control
- Two-phase, non-overlapping clock
- Separate control and datapath subsystems
    - Condition signals
        - ◊ Datapath state to be sensed
        - ◊ Controller will react to these values
        - ◊ List of values affecting control flow (sequencing)
    - Control signals
        - ◊ Evoke operations and transfers in the datapath
        - ◊ Effectively a list of all building block control inputs

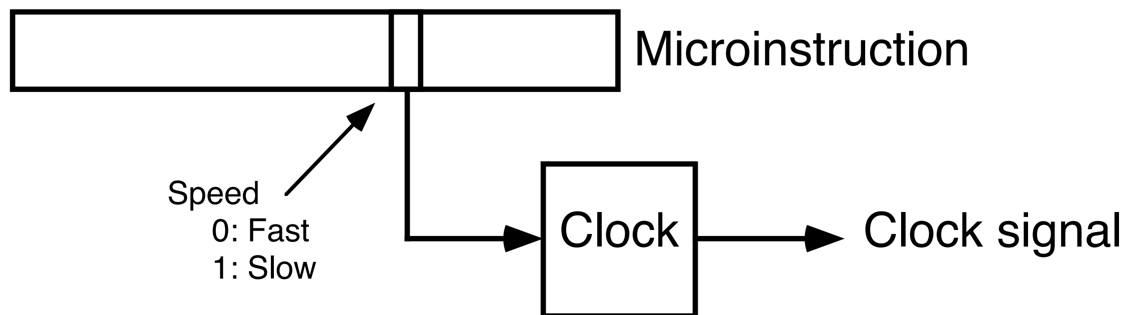


- Microprogrammed control
    - Each control step is a microinstruction
    - A microinstructions is a bundle of control values
- Levels of virtual (abstract) machines

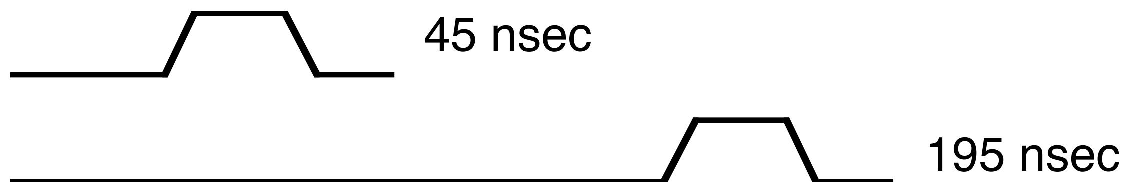


Programmable clock

- Performance limited by slowest component
- Make clock speed selectable
- Choose the fastest speed possible for a transfer



- Two or more speeds



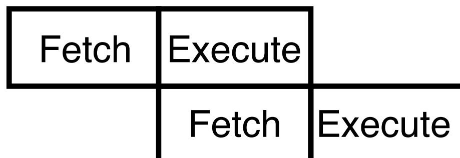
- Fixed speed versus programmable

Operation	Fixed	Programmable
Short operation R ← S	195 nsec	45 nsec
Long operation R ← S Δ R	195 nsec	195 nsec

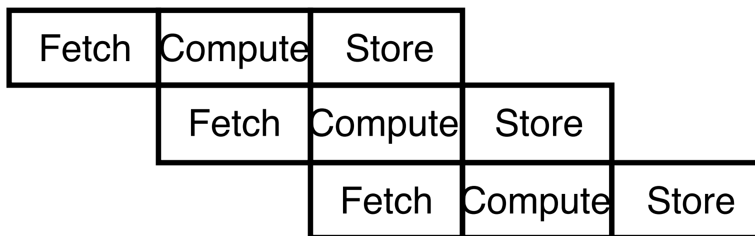
Pipelining

- Perform two or more operations concurrently
- Overlap two or more operations
- Usually later stages of pipe consume results of earlier stages

Instruction lookahead

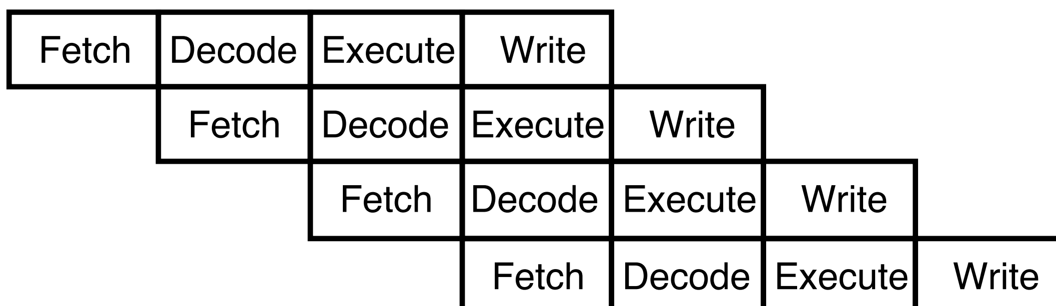


Berkeley RISC machine



- Fetch: Get next instruction in sequence (IP + 1)
- Compute: Read dual port register file and compute result
- Store: Write result to destination register

SPARC pipeline



Pipeline problems

- Hazards
  - Data dependencies
  - Control dependencies
  - Collisions (resource conflict)

- Data dependency
  - Problem: Result is needed before it is stored
  - Example: Berkeley RISC
  - Solution: Detect hazard and forward result

- Control dependency
  - Problem: Branch occurs after prefetch
  - Example: Berkeley RISC and SPARC
  - Solutions
    - Flush and refill pipeline
    - Fill pipeline with no-op instructions
    - Delay the effect of the branch
      - ◊ Execute instructions already in pipe
      - ◊ Branch late
      - ◊ Compile code to use "extra" instruction effectively
      - ◊ Can successfully find work in 90% of the cases

- Collisions (resource conflicts)
  - Two instructions need the same resource
  - Detection and resolution
    - Static
      - ◊ Usually detect at decode stage
      - ◊ Conservative approach - hold until ready
    - Dynamic
      - ◊ Let instruction proceed
      - ◊ Detect and resolve at point of conflict
  - Usage counters
  - Scoreboarding