Processor Data Paths - ALU and Registers

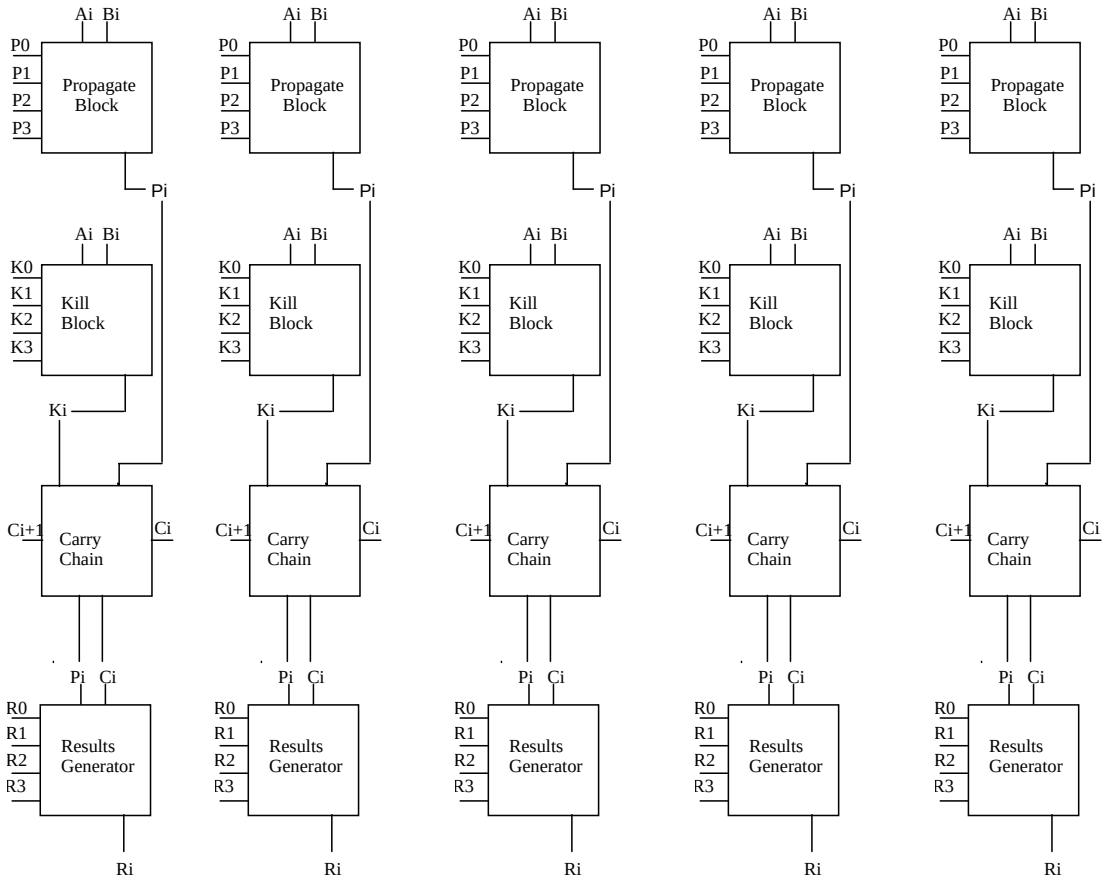
Incorporating the ALU into a Processor Data Path

L4 – Incorporint ALU into data path

- Building up the data path
- Control of the data path

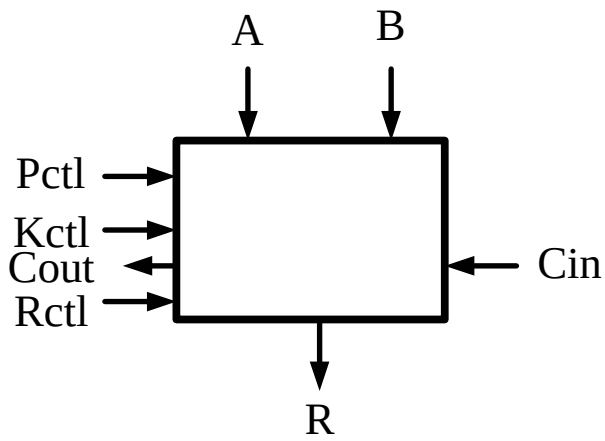
Have our multifunction ALU

- Will now put an interface around the ALU and build up a data path



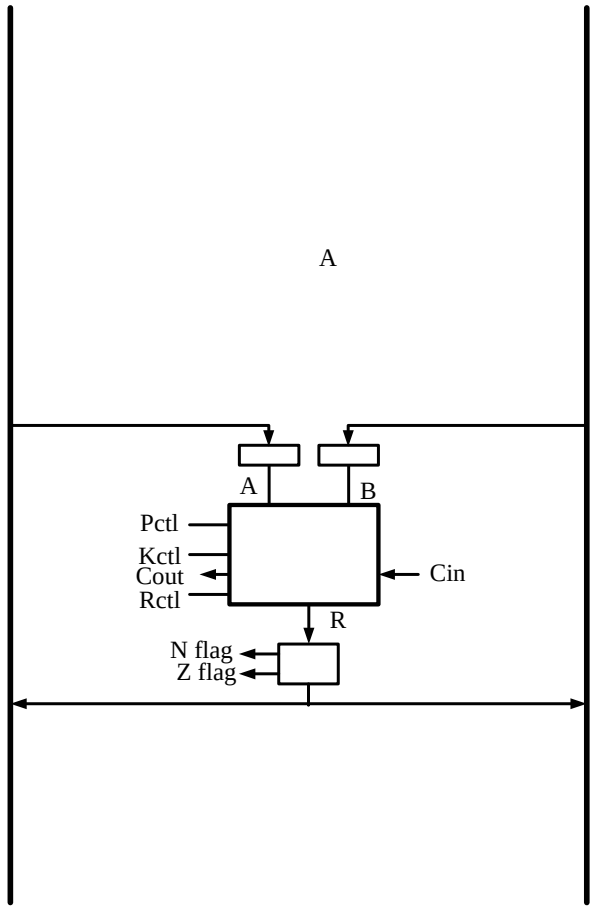
The wrapper

- Establish an interface to the ALU slices
- Have the data inputs
- Have a Carry in
- Have control in
- Produce
  - The result R
  - A carry out



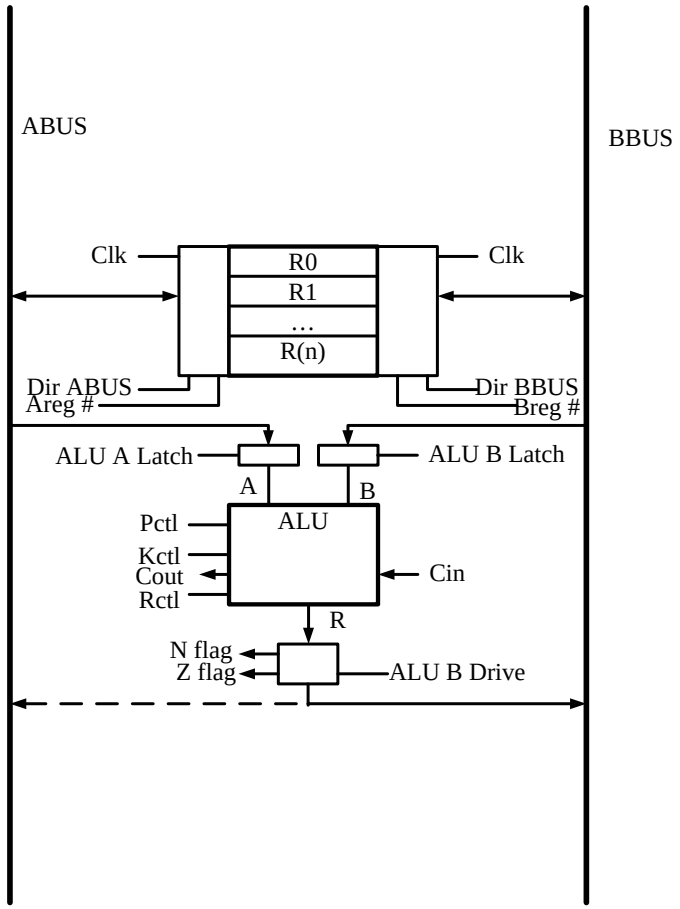
Build up step 1

- Add input and output latches/bus drivers
- Add 2 internal data busses
- Add flag generation
- Inputs & control – A,B,Cin,Pctl, Kctl, Rctl
- Outputs – Cout, R, Nflag, Zflag



Build up step 2

- Now add a general purpose registers
- These are dual ported registers
- Loading or driving of registers on the bus is controlled by the clock
- Busses are tristate

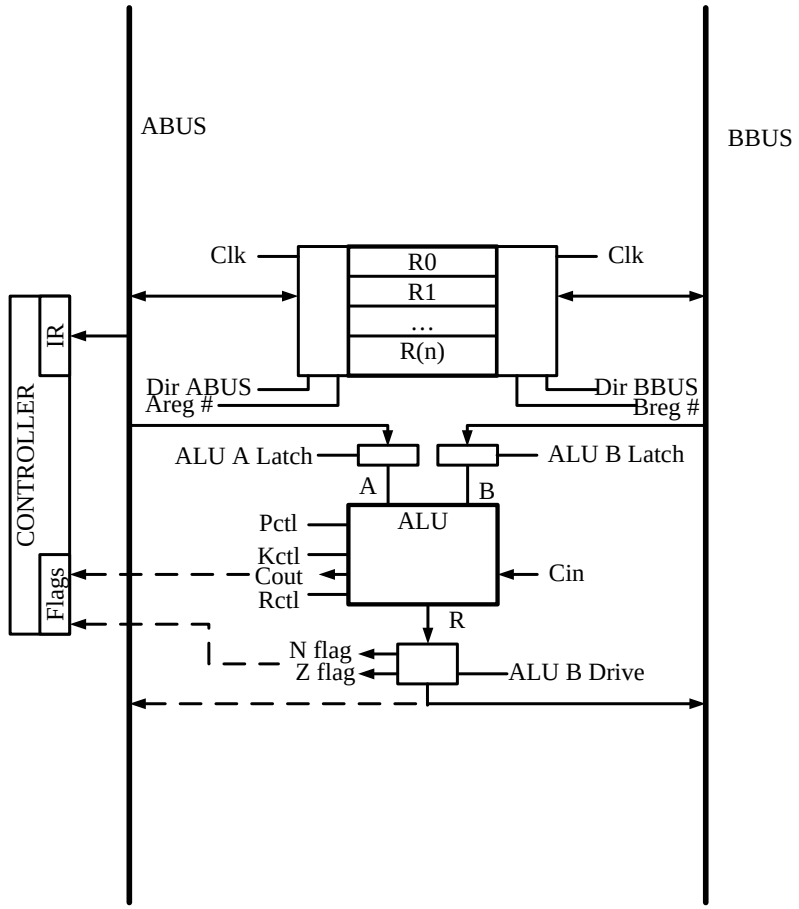


Dual Ported Registers

- Register are dual ported
  - Registers can be loaded from or drive a value onto both the A Bus and B Bus simultaneously
  - Possible combinations on same cycle
    - Drive both the A Bus and B Bus
    - Load from both the A Bus and B Bus
      - Must be different registers!!!!!!!!!
      - Register bank does not check for this – responsibility of the controller
    - Load from one Bus and drive the other

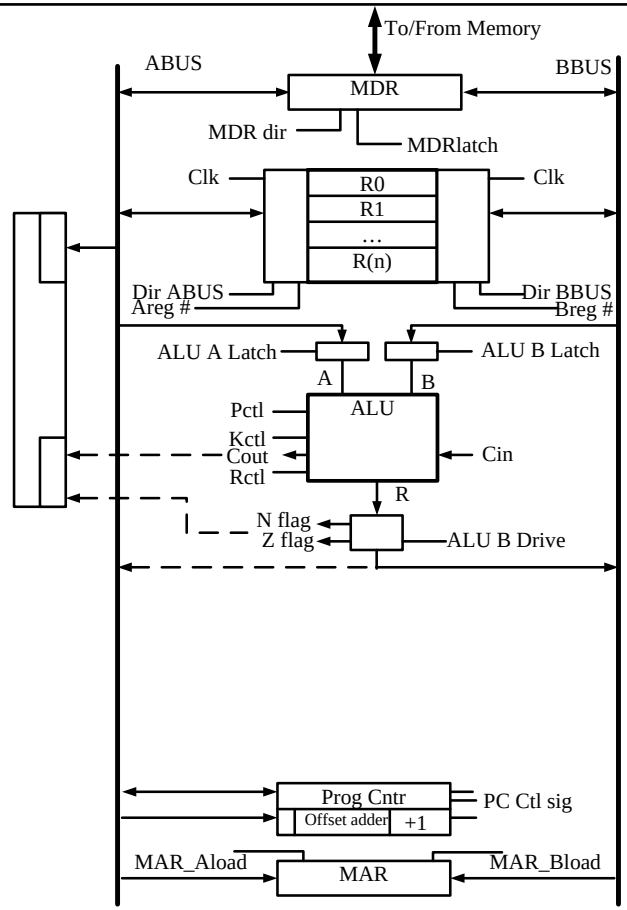
Now add the controller

- Note the Instruction Register and Flags register



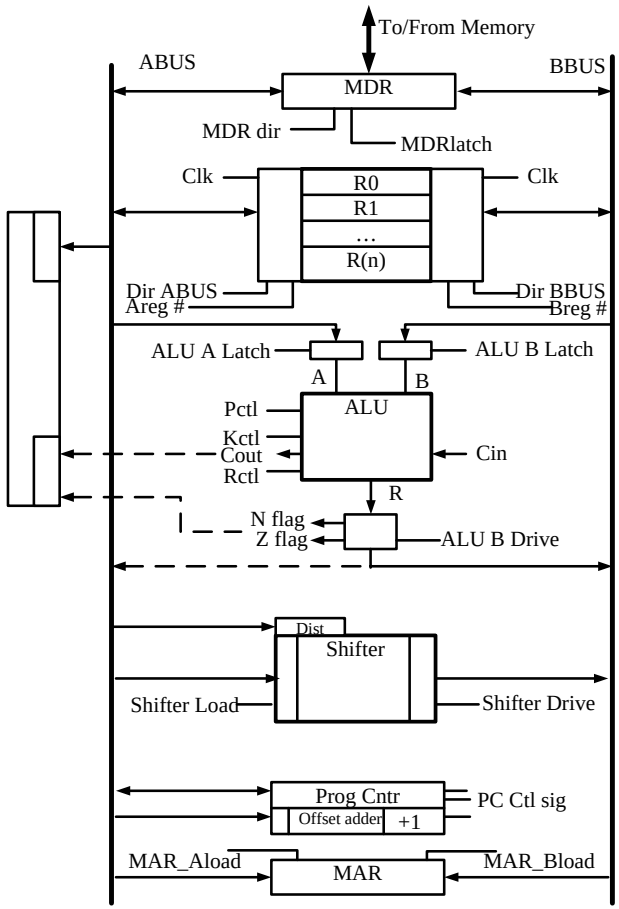
Program Counter and MDR, MAR

- MDR – Memory Data Register
- MAR – Memory Address Register
- PC Register increment and offset integrated into register design



Other Functional Units

- May want to incorporate
  - Shifter
  - Floating Point Units
  - Index addressing registers
  - Etc.



General Operation of a Datapath

- Datapaths are generally synchronous
- Internal clock most likely higher than bus clock
- Typical RTL of datapath
  - Cyc 1: Mem(PC) -> IR
  - PC + 1 -> PC
  - Cyc 2: RS1+RS2 -> ALUout
  - Cyc 3: ALU out -> RS1

Datapath Control Evolution

- Finite State Machine (FSM)
  - Fixed and unvarying action of a datapath
  - FSM implemented with PLA

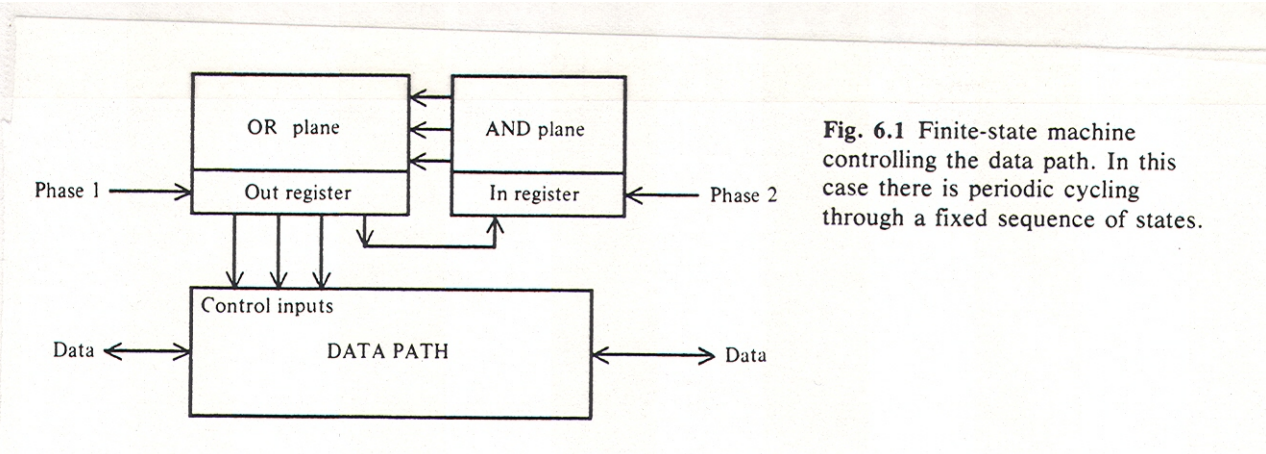


Fig. 6.1 Finite-state machine controlling the data path. In this case there is periodic cycling through a fixed sequence of states.

Use of flags

- Here the current data can affect the action of the controller

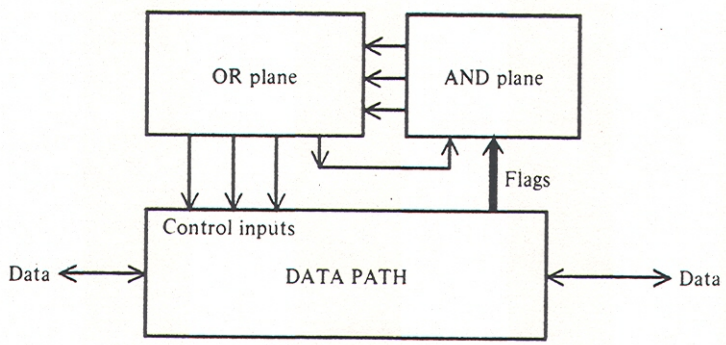


Fig. 6.2 Finite-state machine controlling the data path. In this case the next state can be a function of the previous operation's outcome.

Load flags

- By loading a flags register can use them to affect the controller only at specific points in the “program”

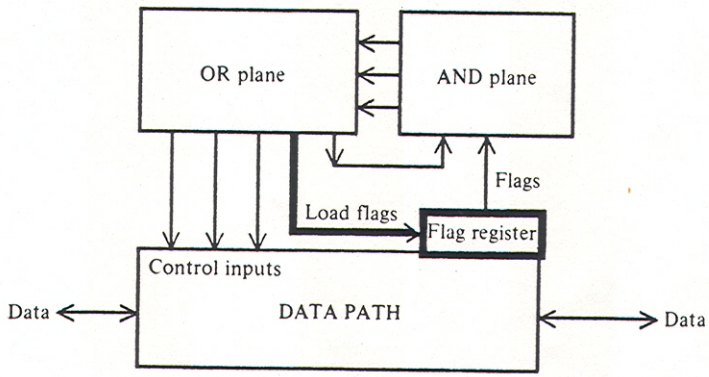


Fig. 6.3 Finite-state machine controlling the data path. In this case a data path operation result may control machine sequencing for a number of later cycles.

Data from memory used

- In addition to flags, data from memory can affect the state and the datapaths action

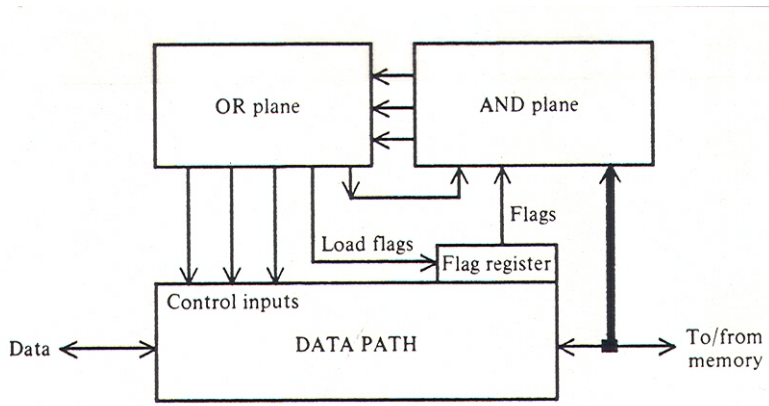


Fig. 6.4 A simple stored-program machine, where data read from a memory can affect machine sequencing.

Instruction Reg

- Have now progressed to a “stored” program

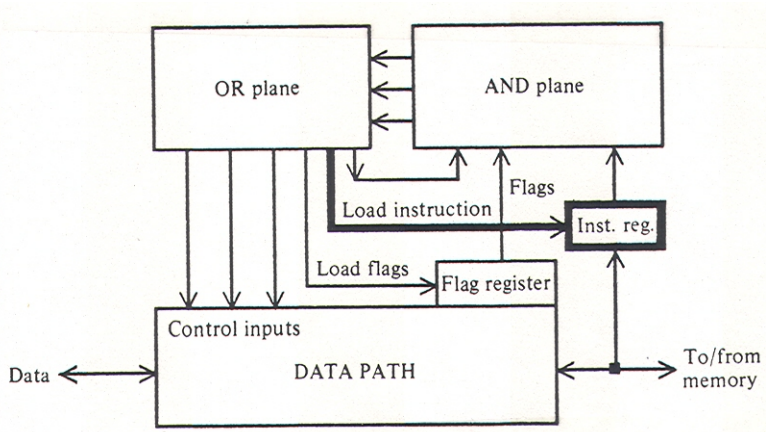


Fig. 6.5 A simple stored-program machine, in this case, augmented by an instruction register.

Alternative form

- Here use a ROM rather than a PLA finite state machine

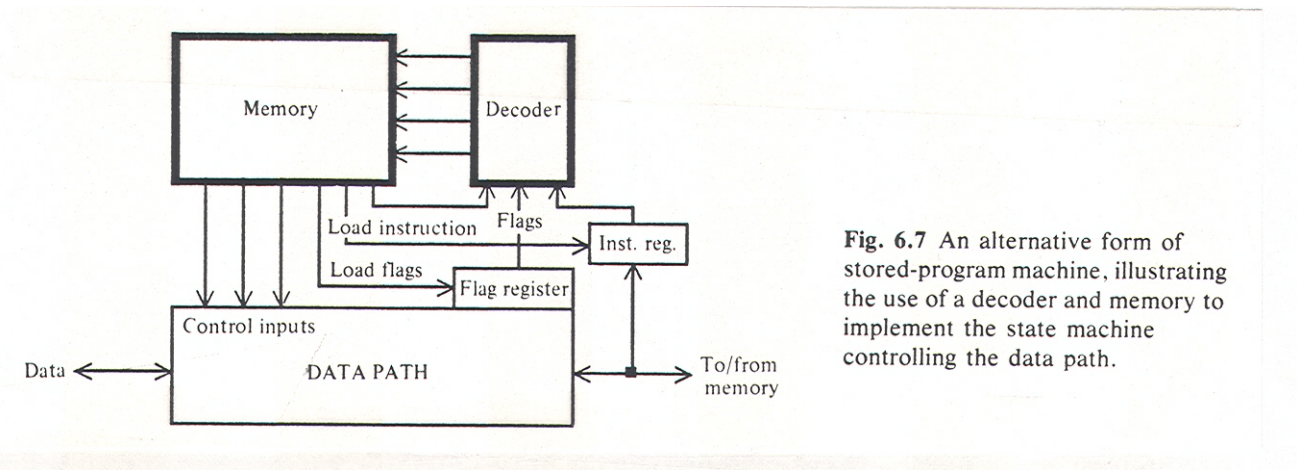


Fig. 6.7 An alternative form of stored-program machine, illustrating the use of a decoder and memory to implement the state machine controlling the data path.

Another alternative

- Use a microprogram counter

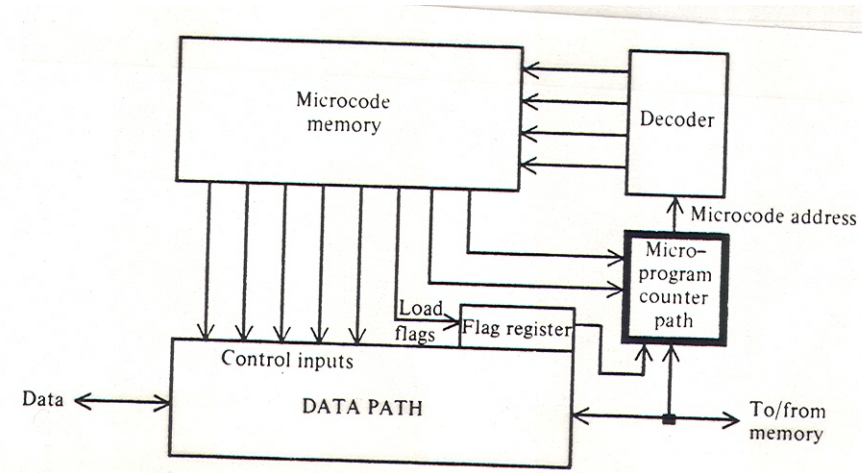


Fig. 6.8 An alternative form of stored-program machine.

A different slice

- Just a different visualization

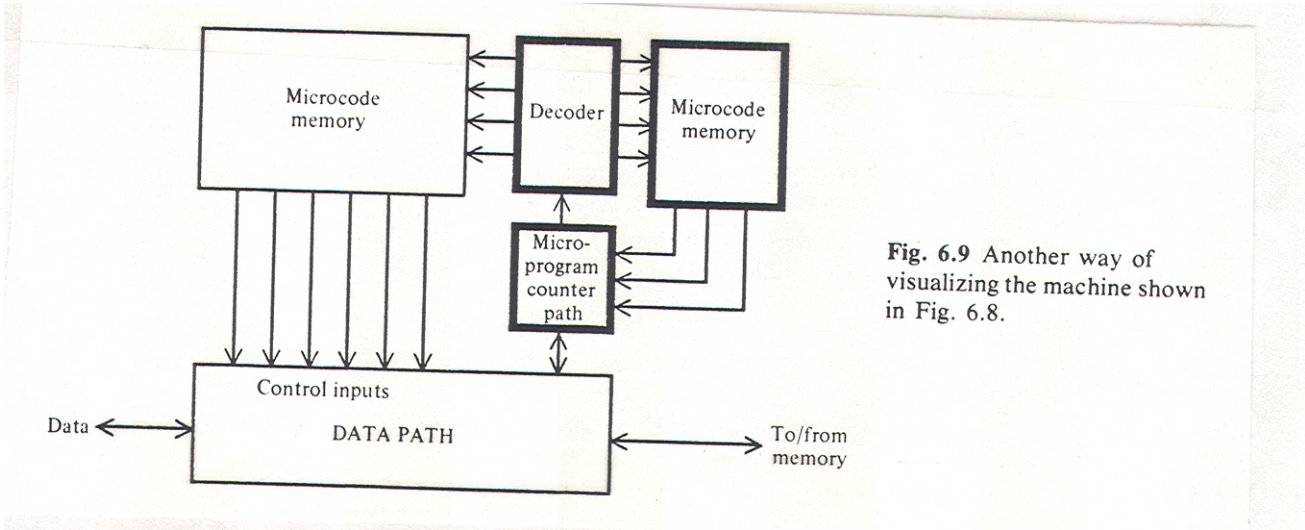


Fig. 6.9 Another way of visualizing the machine shown in Fig. 6.8.

Operation

- Shows steps the controller would go through to execute the instructions

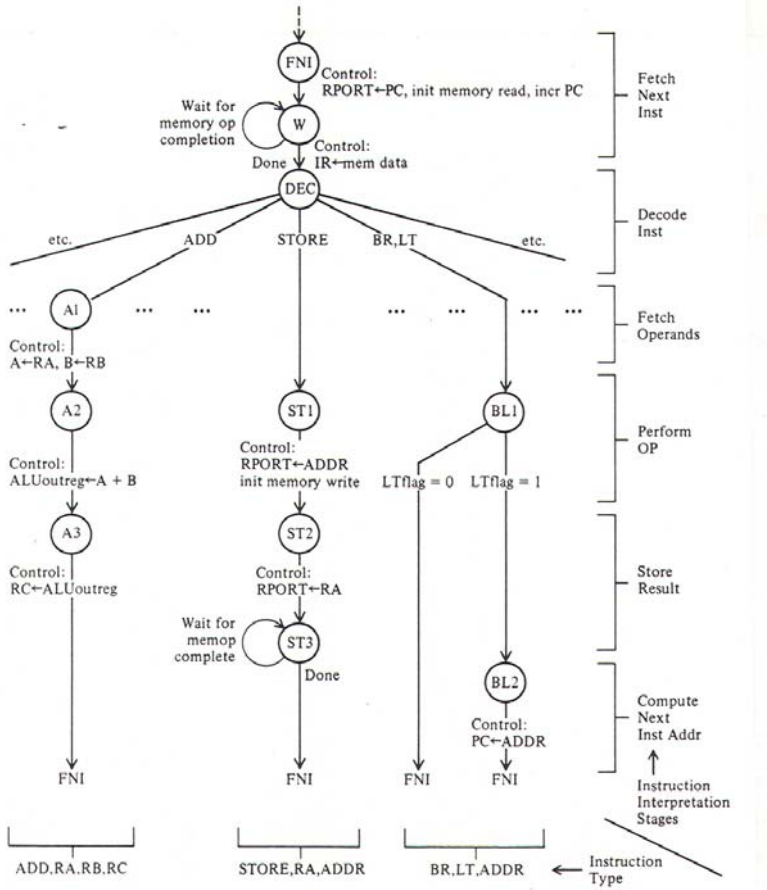


Fig. 6.6 A portion of the state diagram for the controller of a stored-program machine, illustrating some typical instruction interpretation state sequences and their associated control outputs.

Operation Examples – ALU op

Function of sub-sequence	Control [& State] sequence	Comments
Fetch Next Inst:	RPORT ← PC	Place next instr. address in right port.
	read memory	Raise control line to initiate memory read.
	PC ← PC + 1	Increment PC, overlapping incr. with fetch.
	[Y′ = fcn(memop complete)]	Loop here till memory read completes.
	IR ← mem data	Load IR with inst, when read completed.
Decode Instruction:	[Y′ = fcn(IR)]	Set machine state as fcn of instruction.
Fetch Operands:	A ← R7	Load ALU input registers with operands.
	B ← R2	
Perform Operation:	ALUoutreg ← A + B	Add A and B, store in output register.
Store Result:	R5 ← ALUoutreg	Send result address to R5.
	[Y′ = FNI]	Inst. not a branch, so simply return to FNI state.

Store result back to memory

Function of sub-sequence	Control [& State] sequence	Comments
Fetch Next Inst:	RPORT ← PC	Place next instr. address in right port.
	read memory	Raise control line to initiate memory read.
	PC ← PC+1	Increment PC.
	[Y′ = fcn(memop complete)]	Loop here till memory read completes.
	IR ← mem data	Load IR with inst, when read completed.
Decode Instruction:	[Y′ = fcn(IR)]	Set machine state as fcn of instruction.
Perform Operation:	RPORT ← IR(ADDRESS)	Send the contents of IR address field to memory.
	write memory	Raise write control line to init. memory write.
Store Result:	RPORT ← R3	Place data in right output port.
	[Y′ = fcn(memop complete)]	Loop here till memory write completes.
	[Y′ = FNI]	Inst. not a branch, so simply return to FNI state.