Chapter #11: Computer Organization

Contemporary Logic Design

Randy H. Katz
University of California, Berkeley

July 1993

Motivation

- Computer Design as an application of digital logic design procedure
- Computer = Processing Unit + Memory System
- Processing Unit = Control + Datapath
- Control = Finite State Machine

  Inputs = Machine Instruction, Datapath Conditions

  Outputs = Register Transfer Control Signals

  Instruction Interpretation = Instruction Fetch, Decode, Execute
- Datapath = Functional Units + Registers

  Functional Units = ALU, Multipliers, Dividers, etc.

  Registers = Program Counter, Shifters, Storage Registers

Chapter Overview

Design of Datapaths and Processor Control Units

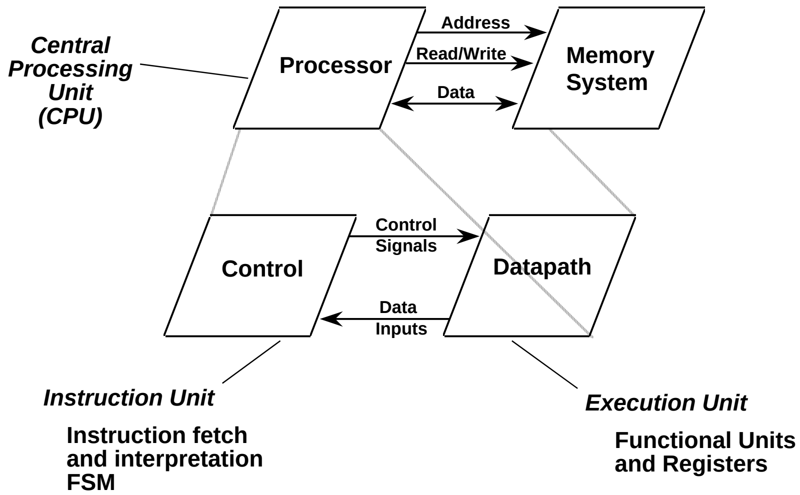
- Datapath interconnection strategies:

  Point-to-Point, Single Bus, Multiple Busses

- Structure of the State Diagram/ASM Chart to describe controller FSM

Structure of a Computer

Block Diagram View



Structure of a Computer

Example of Instruction Sequencing

Instruction: Add Rx to Ry and place result in Rz

Step 1: *Fetch the Add instruction from Memory to Instruction Reg*

Step 2: *Decode Instruction*

Instruction in IR is an ADD

Source operands are Rx, Ry

Destination operand is Rz

Step 3: *Execute Instruction*

Move Rx, Ry to ALU

Set up ALU to perform ADD function

ADD Rx to Ry

Move ALU result to Rz

Structure of a Computer

Instruction Types

- Data Manipulation

  Add, Subtract, etc.

- Data Staging

  Load/Store data to/from memory

  Register-to-register move

- Control

  Conditional/unconditional branches

  subroutine call and return

Structure of a Computer

Control

Elements of the Control Unit (aka Instruction Unit):

Standard FSM things:

State Register

Next State Logic

Output Logic (datapath control signaling)

Plus Additional "Control" Registers:

Instruction Register (IR)

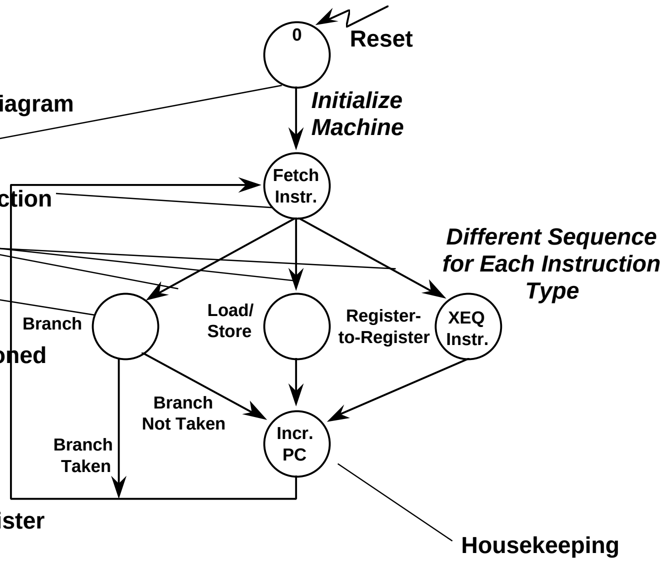
Program Counter (PC)

Structure of a Computer

Control

Control State Diagram

- Reset
- Fetch Instruction
- Decode
- Execute



Instructions partitioned into three classes:

- Branch
- Load/Store
- Register-to-Register

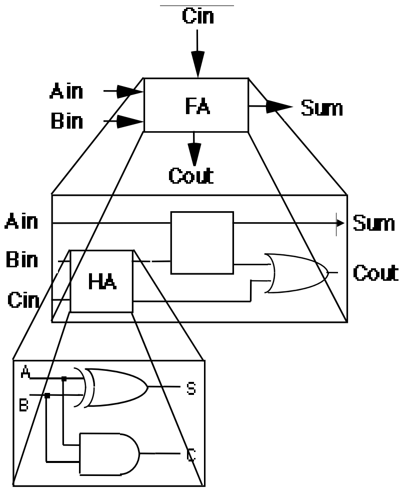
Structure of a Computer

Datapath

Arithmetic Circuits constructed in hierarchical and iterative fashion

Each bit in datapath is functionally identical

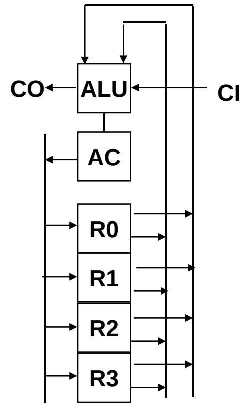
4-bit
8-bit
16-bit
32-bit
Datapaths



Hierarchical Construction of Full Adder

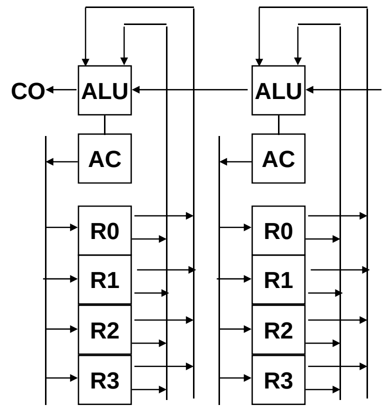
Structure of a Computer

Datapath



1 bit wide

Bit Slice Concept



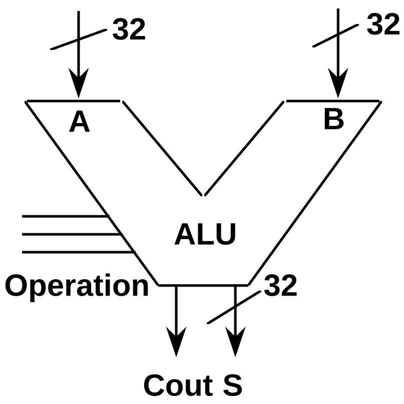
2 bits wide

iterate to build n-bit wide datapaths

Structure of a Computer

Datapath

ALU Block Diagram



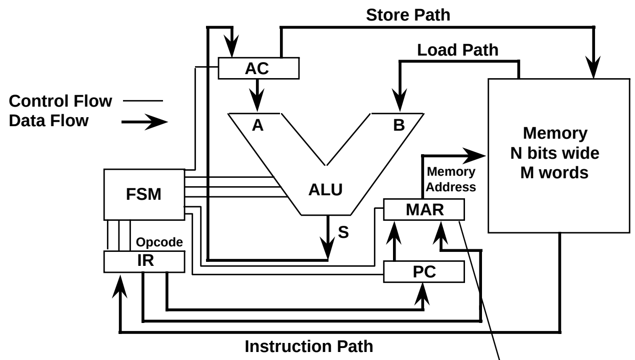
Structure of a Computer

Block Diagram/Register Transfer View

Single Accumulator Machine

AC := AC <op> Mem

"single address instructions"
AC implicit operand



Arrowed Lines represent dataflows

others are control flows

Memory Address Register

Hold address during memory accesses

Structure of a Computer

Block Diagram/Register Transfer View

Placement of Data and Instructions in Memory:

- Data and instructions mixed in memory: Princeton Architecture
- Data and instructions in separate memory: Harvard Architecture

Princeton architecture simpler to implement

Harvard architecture has certain performance advantages:

overlap instruction fetch with operand fetch

We assume the more common Princeton architecture throughout

Structure of a Computer

Block Diagram/Register Transfer View

Trace an instruction: AC := AC + Mem<address>

1. Instruction Fetch:

   Move PC to MAR

   Initiate a memory read sequence

   Move data from memory to IR

2. Instruction Decode:

   Op code bits of IR are input to control FSM

   Rest of IR bits encode the operand address

Structure of a Computer

Block Diagram/Register Transfer View

Trace an instruction: AC := AC + Mem<address>

3. Operand Fetch:

   Move operand address from IR to MAR

   Initiate a memory read sequence

4. Instruction Execute:

   Data available on load path

   Move data to ALU input

   Configure ALU to perform ADD operation

   Move S result to AC

5. Housekeeping:

   Increment PC to point at next instruction

Structure of a Computer

Block Diagram/Register Transfer View

Control: Transfer data from one register to another
Assert appropriate control signals

Register transfer notation — *Register to Register moves*

Ifetch:	PC → MAR; Memory Read; Memory → IR;	-- move PC to MAR -- assert Memory READ signal -- load IR from Memory
Instruction Decode:	IF IR<op code> = ADD_FROM_MEMORY THEN	
Instruction Execution:	IR<addr> → MAR; Memory Read;	-- move operand addr to MAR -- assert Memory READ signal
	Memory → ALU B; AC → ALU A; ALU ADD;	-- gate Memory to ALU B -- gate AC to ALU A -- instruct ALU to perform ADD
	ALU S → AC;	-- gate ALU result to AC
Assert Control Signal	PC+1;	-- increment PC

Structure of a Computer

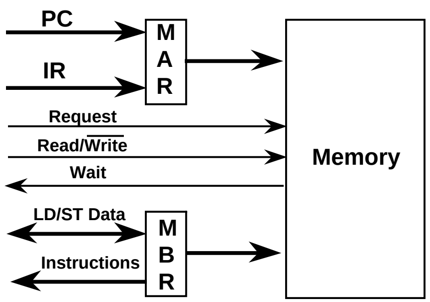
Memory Interface

More Realistic Block Diagram:

Issue memory request

Is it a read or a write?

Memory asks CPU to wait



Decouple memory system from internal processor operation

Memory Buffer Register

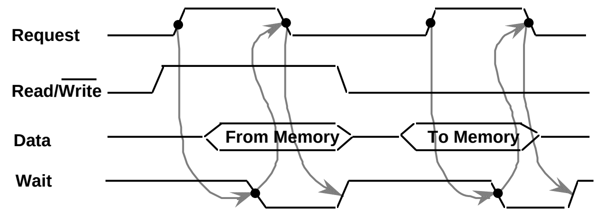
Structure of a Computer

Memory Interface

No common clock between CPU and memory

Follow asynchronous 4-cycle handshake request/wait ($\overline{ack}$) protocol

1. Request Asserted
2. Wait Unasserted
3. Request Unasserted
4. Wait Asserted



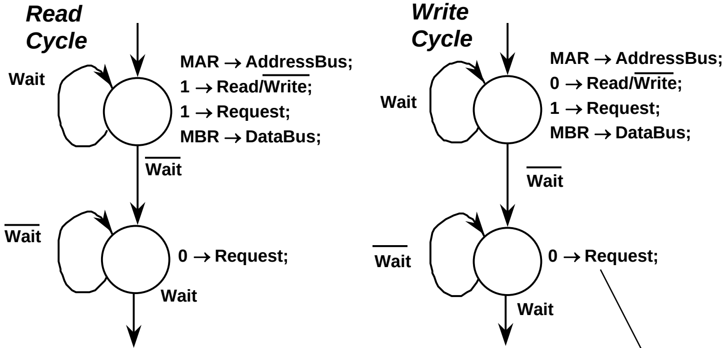
Memory cannot make request unless Wait signal is asserted

Hi-to-Lo transition on Wait implies that data is ready (read) or data has been latched by memory (write)

Structure of a Computer

Memory Interface

State Diagram Fragments for Read/Write Cycles



State 1: drive address bus
assert read request
catch data into MBR

State 2: unassert request
hold in state until Wait reasserted

Normal Convention:

If register transfer op NOT asserted, it need not be mentioned in state diagram

Structure of a Computer

I/O Interface

Memory-Mapped I/O

- I/O devices share the memory address space
- Control registers manipulated just like memory word
- Read/write register to initiate I/O operation

Polling

- Programs periodically checks whether I/O has completed

Interrupts

- Device signals CPU when operation is complete
- Software must take over to handle the data transfers from the device
- Check for interrupt pending before fetching next instruction
- Save PC & vector to special memory location for next instruction
- Instruction set includes a "return from interrupt" instruction

Bussing Strategies

Register-to-Register Coummunications

- Point-to-point
- Single shared bus
- Multiple special purpose busses

Tradeoffs between datapath/control complexity and amount of parallelism supported by the hardware

Case study:

Four general purpose registers that must be able to exchange their contents

Swap instruction must be supported:

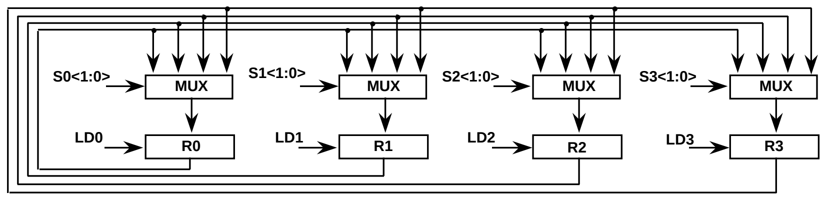
SWAP(Ri, Rj)

Ri → Rj;

Rj → Ri;

Bussing Schemes

Point-to-Point Connection Scheme



Four registers interconnected via 4:1 Mux's and point-to-point connections

- Edge-triggered N bit registers controlled by LDi signals
- N x 4:1 MUXes per register, controlled by Si<1:0> signals

Bussing Schemes

Point-to-point Connections

Example:

Register transfers R1 → R0 and R2 → R3

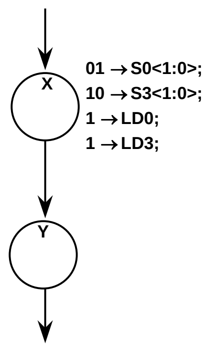
Register transfer operations:

01 → S0<1:0>;	Enable path from R1 to R0
10 → S3<1:0>;	Enable path from R2 to R3
1 → LD0;	Assert load for R0
1 → LD3;	Assert load for R3

Bussing Schemes

Point-to-point Connections

When control signals are asserted and when they take place:



Moore Machine
State Diagram

Enter state X:
Multiplexor control signals asserted
R1 outputs arrive at R0 inputs
R2 outputs arrive at R3 inputs

LD signals asserted
Do not take effect until next rising clock

On entering state Y:
LD signals are synchronous and take effect at the same time as the state transition!

Bussing Schemes

Point-to-point connections

Implementation of Register SWAP operation

SWAP(R1, R2):

01 → S2<1:0>;
10 → S1<1:0>; — Establish connection paths
1 → LD2;
1 → LD1; — Swap takes place at next state transition

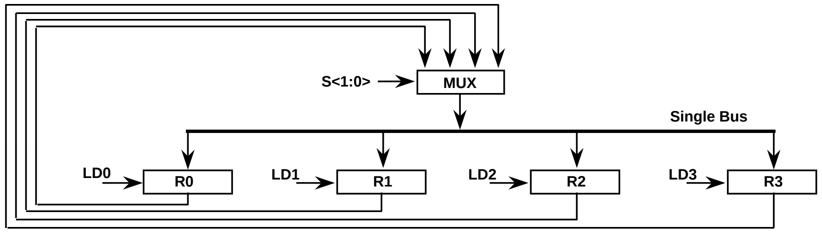
Point-to-Point Scheme Plusses and Minuses:

+ transfer a new value into each of the four registers at same time
+ register swap implemented in a single control state

- 5 gates to implement 4:1 MUX
  32 bit wide datapath implies 32 x 5 x 4 registers
  = 640 gates!
  very expensive implementation

Bussing Strategies

Single Bus Interconnection



- per register MUX block replaced by single block
- 25% hardware cost of previous alternative
- shared set of pathways is called a BUS

Single bus becomes a critical resource --
used by only one transfer at a time

Bussing Strategies

Single Bus Interconnection

Example: R1 → R0 and R2 → R3

State X: (R1 → R0)
01 → S<1:0>;
1 → LD0;

State Y: (R2 → R3)
10 → S<1:0>;
1 → LD3;

Datapath no longer supports two simultaneous transfers!
Thus two control states are required to perform the transfers

Bussing Strategies

Single Bus Interconnection

SWAP Operation

A special TEMP register must be introduced ("Register 4")
MUX's become 5:1 rather than 4:1

State X: (R1 → R4)
001 → S<2:0>;
1 → LD4;

Three states are required rather than one!

plus extra register and wider mux

More control states because this datapath supports less parallel activity

State Y: (R2 → R1)
010 → S<2:0>;
1 → LD1;

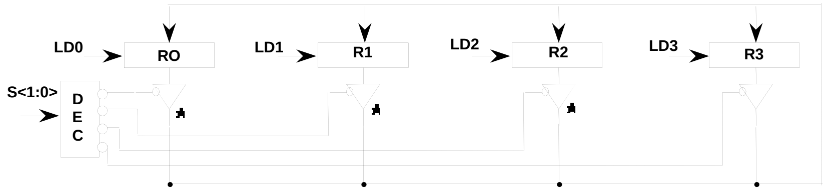
Engineering choices made based on how frequently multiple transfers take place at the same time

State Z: (R4 → R2)
100 → S<2:0>
1 → LD2;

Bussing Strategies

Alternatives to Multiplexors

Tri-state buffers as an interconnection scheme



Only one register's contents gated to shared bus at a time

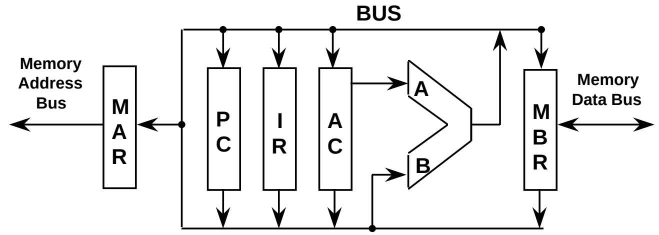
Bussing Strategies

Multiple Busses

Real datapaths are a compromise between the two extremes

Register Transfer Diagram

Single Bus Design



Register transfer operations:

PC → BUS
IR → BUS
AC → BUS
MBR → BUS
ALU Result → BUS

BUS → PC
BUS → IR
BUS → AC
BUS → MBR
BUS → ALU B
BUS → MAR

AC → ALU A
("hardwired")

Bussing Strategies

Multiple Busses

Example Register Transfer for Single Bus Design

Instruction Interpretation for "ADD Mem[X]"

Fetch Operand
Cycle 1: IR<operand address> → BUS;
BUS → MAR;

Cycle 2: Memory Read;
Databus → MBR;

Perform ADD
Cycle 3: MBR → BUS;
BUS → ALU B;
AC → ALU A;
ADD;

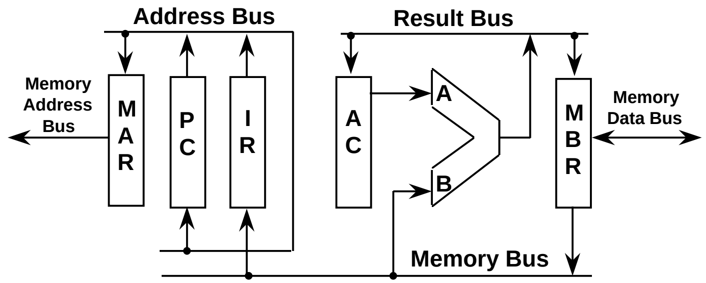
Write Result
Cycle 4: ALU Result → BUS;
BUS → AC;

Requires latch for ALU Result

Bussing Strategies

Multiple Busses

Three Bus Design -- Supports more parallelism



Single bus replaced by three busses:

Memory Bus (MBUS)
Result Bus (RBUS)
Address Bus (ABUS)

Bussing Strategies

Multiple Busses

Instruction Interpretation for "ADD Mem[X]"

Fetch Operand

Cycle 1: IR<operand address> → ABUS;
ABUS → MAR;

Cycle 2: Memory Read;
Databus → MBR;

Perform ADD

Cycle 3: MBR → MBUS;
MBUS → ALU B;
AC → ALU A;
ADD;

Implemented in three cycles rather than four

Write Result

ALU Result → RBUS;
RBUS → AC;

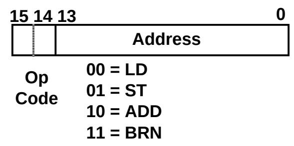
Advantage of separate ABUS:
overlap PC → MAR with instruction execution

Finite State Machines for Simple CPUs

State Diagram and Datapath Derivation

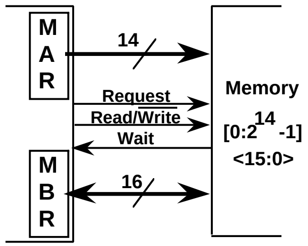
Processor Specification:

Instruction Format:



Load from memory: Mem[XXX] → AC;
Store to memory: AC → Mem[XXX];
Add from memory: AC + Mem[XXX] → AC;
Branch if accumulator is negative: AC < 0 ⇒ XXX → PC;

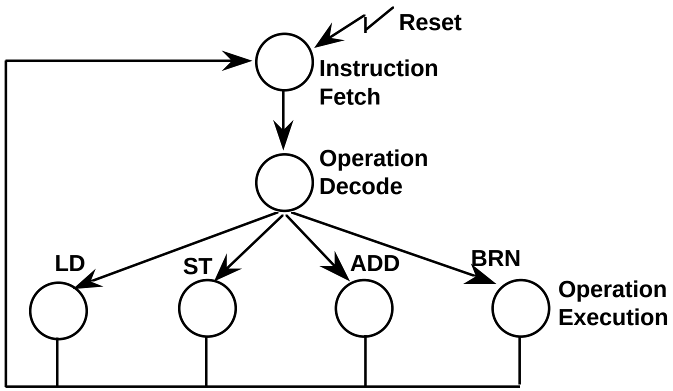
Memory Interface:



Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

First pass state diagram:



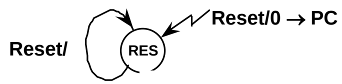
Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Assume Synchronous Mealy Machine:

Transitions associated with arcs rather than states

Reset State (State 0) and Instruction Fetch Sequence



On Reset:
- zero the PC
- Mem Request unasserted
- Mem asserts Wait signal

Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

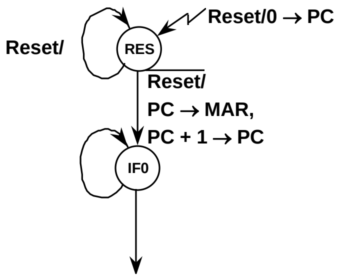
Assume Synchronous Mealy Machine:

Transitions associated with arcs rather than states

Reset State (State 0) and Instruction Fetch Sequence

On Reset:
- zero the PC
- Mem Request unasserted
- Mem asserts Wait signal

Instruction Fetch:
- issue read request
- 4 cycle handshake on Wait signal



Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

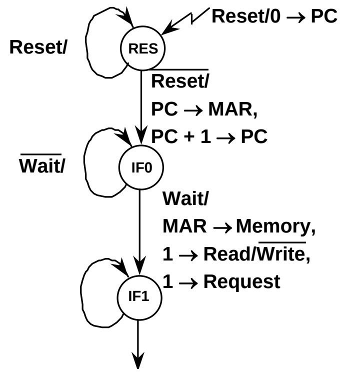
Assume Synchronous Mealy Machine:

Transitions associated with arcs rather than states

Reset State (State 0) and Instruction Fetch Sequence

On Reset:
- zero the PC
- Mem Request unasserted
- Mem asserts Wait signal

Instruction Fetch:
- issue read request
- 4 cycle handshake on Wait signal



Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

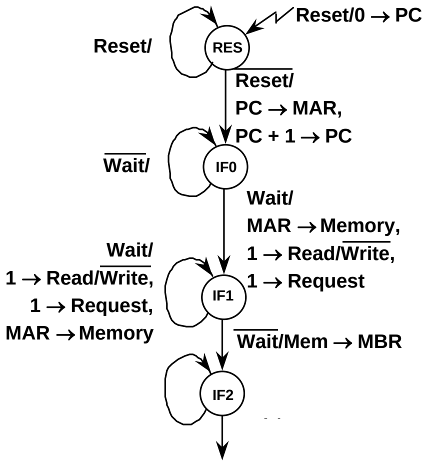
Assume Synchronous Mealy Machine:

Transitions associated with arcs rather than states

Reset State (State 0) and Instruction Fetch Sequence

On Reset:
- zero the PC
- Mem Request unasserted
- Mem asserts Wait signal

Instruction Fetch:
- issue read request
- 4 cycle handshake on Wait signal



Note: No explicit mention of the busses being used to implement register transfers!

Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

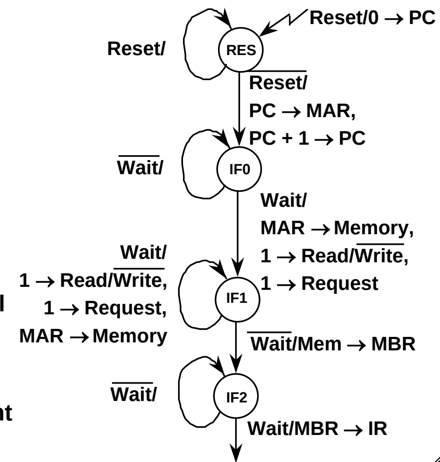
Assume Synchronous Mealy Machine:

Transitions associated with arcs rather than states

Reset State (State 0) and Instruction Fetch Sequence

On Reset:
- zero the PC
- Mem Request unasserted
- Mem asserts Wait signal

Instruction Fetch:
- issue read request
- 4 cycle handshake on Wait signal

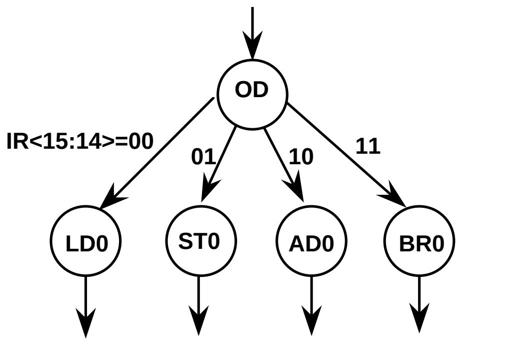


Note: No explicit mention of the busses being used to implement register transfers!

Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Operation Decode State



Four Way Next State Branch based on opcode bits

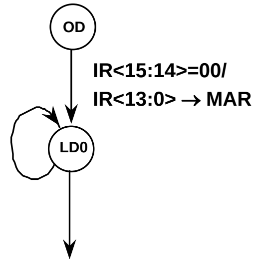
Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Execution Sequences

Load Sequence

like IFetch, except that operand address comes from IR and data should be loaded into AC



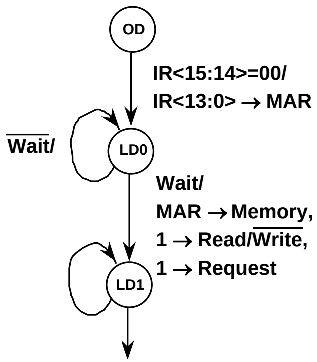
Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Execution Sequences

Load Sequence

like IFetch, except that operand address comes from IR and data should be loaded into AC



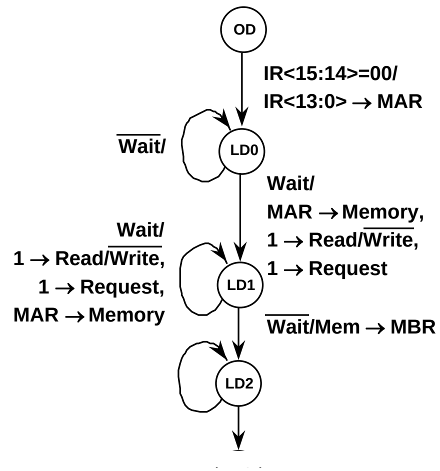
Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Execution Sequences

Load Sequence

like IFetch, except that operand address comes from IR and data should be loaded into AC



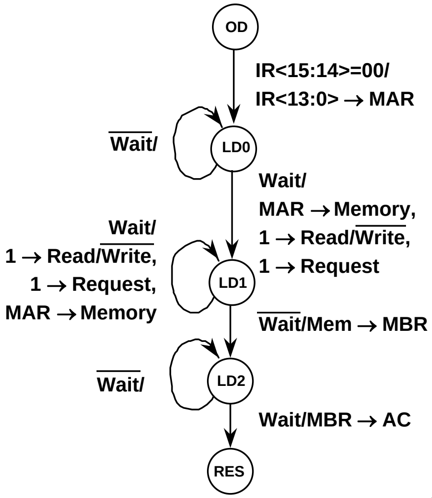
Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Execution Sequences

Load Sequence

like IFetch, except that operand address comes from IR and data should be loaded into AC

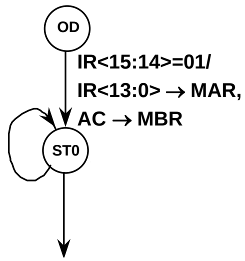


Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Store Execution Sequence

Memory write sequence

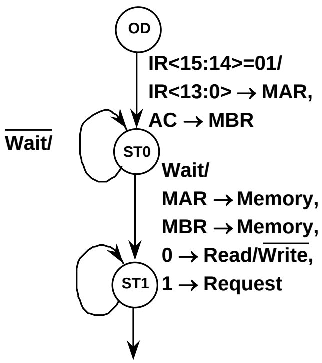


Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Store Execution Sequence

Memory write sequence

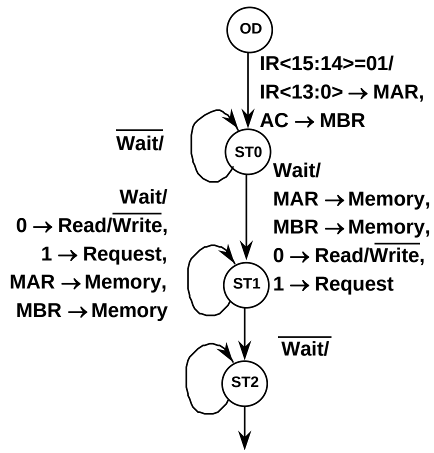


Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Store Execution Sequence

Memory write sequence

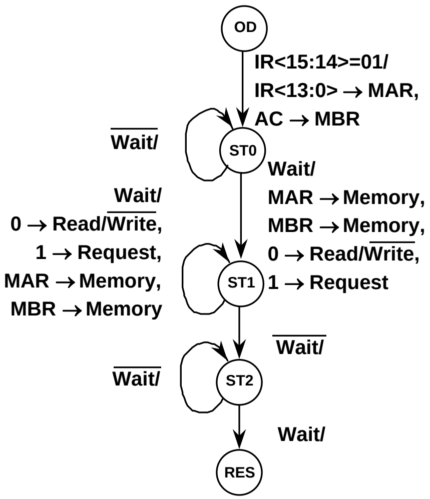


Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Store Execution Sequence

Memory write sequence

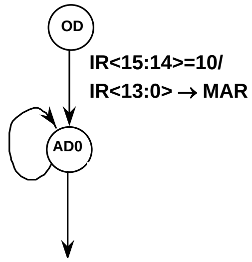


Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Add Execution Sequence

Similar to Load sequence
Add MBR, AC rather than simply transfer MBR to AC

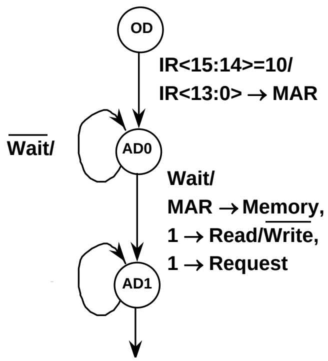


Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Add Execution Sequence

Similar to Load sequence
Add MBR, AC rather than simply transfer MBR to AC

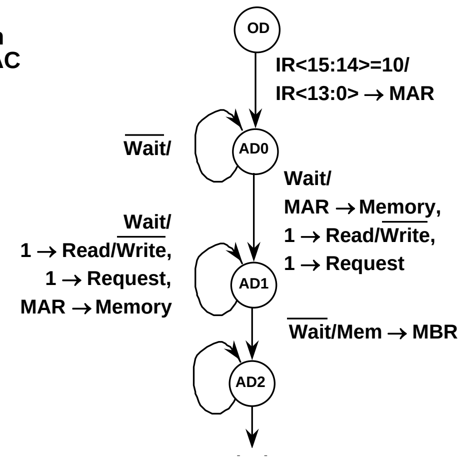


Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Add Execution Sequence

Similar to Load sequence
Add MBR, AC rather than simply transfer MBR to AC

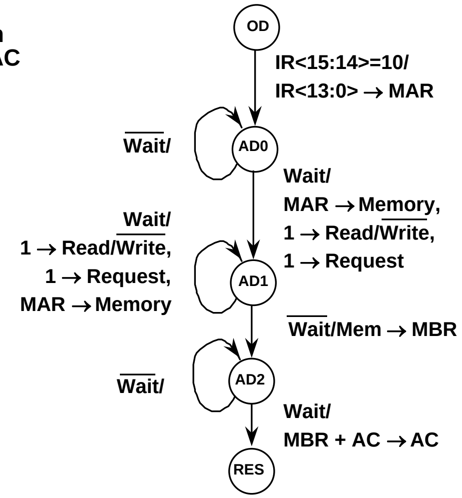


Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

Add Execution Sequence

Similar to Load sequence
Add MBR, AC rather than
simply transfer MBR to AC

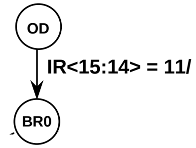


Finite State Machines for Simple CPUs

Contemporary Logic Design
Computer Organization

Deriving the State Diagram and Datapath

Branch Execution Sequence

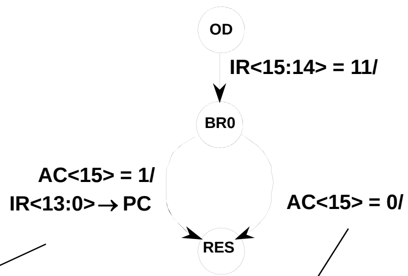


Finite State Machines for Simple CPUs

Contemporary Logic Design
Computer Organization

Deriving the State Diagram and Datapath

Branch Execution Sequence



Replace PC with
Operand Address if
AC < 0

Otherwise, do nothing

Finite State Machines for Simple CPUs

Contemporary Logic Design
Computer Organization

Deriving the State Diagram and Datapath

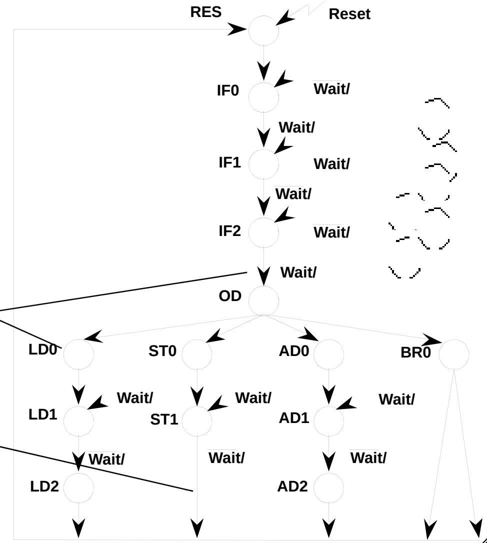
Revised/Complete State Diagram

Simplify Wait Looping

Eliminate some Wait states

At this point, Wait must be
asserted, so why loop on
$\overline{\text{Wait}}$?

Why loop on $\overline{\text{Wait}}$ when
resync will take place at
state IF0?



Finite State Machines for Simple CPUs

Deriving the State Diagram and Datapath

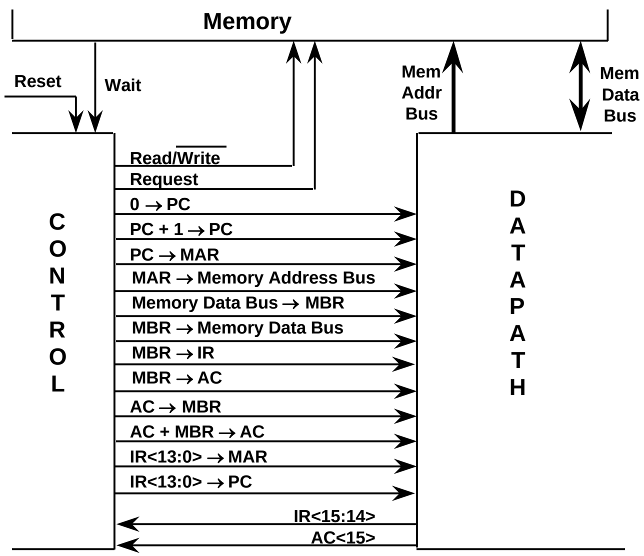
State Machines Inputs and Outputs so far:

Inputs:
- Reset
- Wait
- IR<15:14>
- AC<15>

Outputs:
- 0 → PC
- PC + 1 → PC
- PC → MAR
- MAR → Memory Address Bus
- Memory Data Bus → MBR
- MBR → Memory Data Bus
- MBR → IR
- MBR → AC
- AC → MBR
- AC + MBR → AC
- IR<13:0> → MAR
- IR<13:0> → PC
- 1 → Read/Write
- 0 → Read/Write
- 1 → Request

Finite State Machines for Simple CPUs

Processor Signal Flow

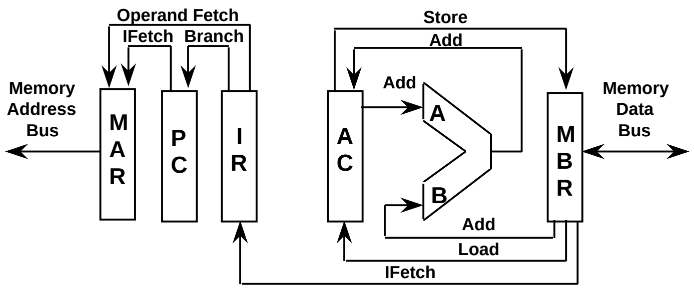


Finite State Machines for Simple CPUs

Mapping onto Datapath Control

Specification so far is independent of bussing strategy

Implied transfers:



This is the point-to-point connection scheme

Finite State Machines for Simple CPUs

Mapping onto Datapath Operations

Observe that instruction fetch and operand fetch take place at different times

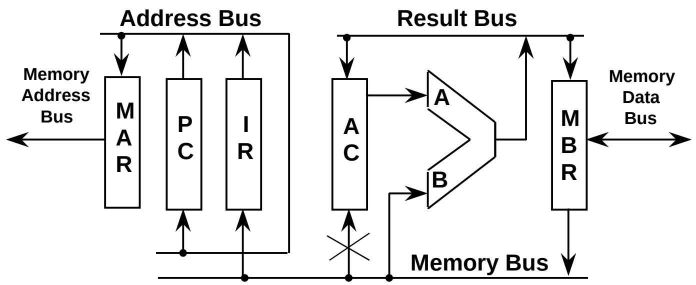
This implies that IR, PC, and MAR transfers can be implemented by single bus (Address Bus)

Combine MBR, IR, ALU B, and AC connections (Memory Bus)

Combine ALU, AC, and MBR connections (Result Bus)

Three bus architecture:
AC + MBR → AC implemented in single state

Mapping onto Datapath Operations



AC has two inputs, RBUS and MBUS
(Other registers except MBR have single input and output)

Dual ported configuration is more complex

Better idea: reuse existing paths were possible
MBR → AC transfer implemented by PASS B ALU operation

Mapping onto Datapath Operations

Detailed implementation of register transfer operations

More detailed control operations are called *microoperations*

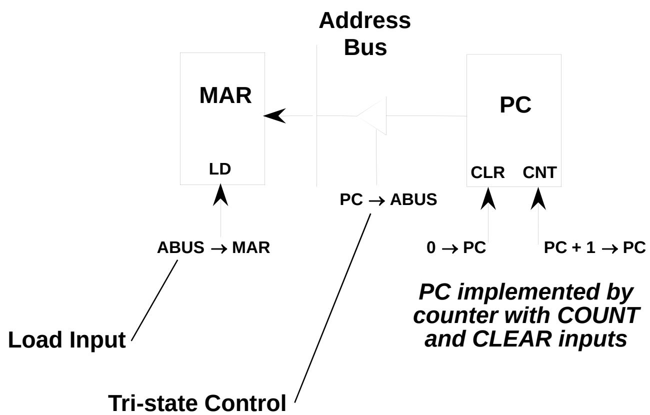
One register transfer operation = several microoperations

Some operations directly implemented by functional units:
e.g., ADD, Pass B, 0 → PC, PC + 1 → PC

Some operations require multiple control operations:

e.g., PC → MAR implemented as
PC → ABUS and ABUS → MAR

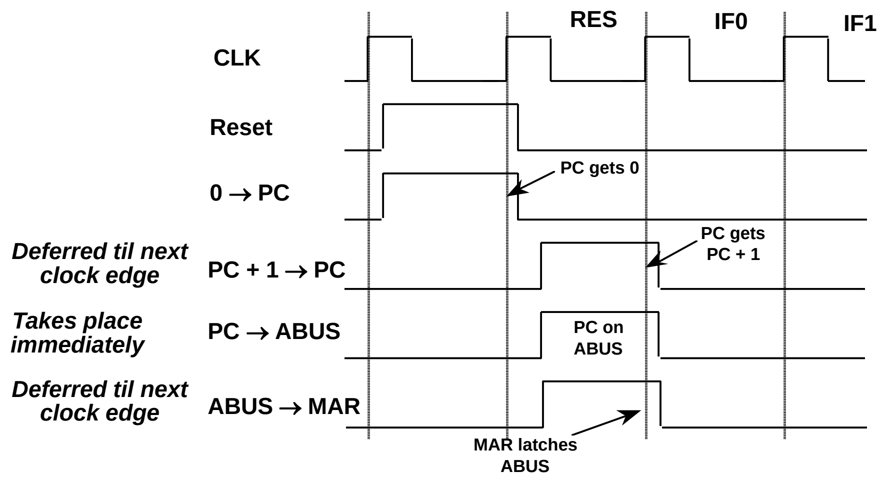
Mapping onto Datapath Operations



PC implemented by counter with COUNT and CLEAR inputs

Mapping onto Datapath Operations

Timing of State Changes and Microoperations



Finite State Machines for Simple CPUs

Mapping onto Datapath Operations

Relationship between register transfer and microoperations:

Register Transfer	Microoperations
0 → PC	0 → PC (delayed);
PC + 1 → PC	PC + 1 → PC (delayed);
PC → MAR	PC → ABUS (immediate),
	ABUS → MAR (delayed);
MAR → Address Bus	MAR → Address Bus (immediate);
Data Bus → MBR	Data Bus → MBR (delayed);
MBR → Data Bus	MBR → Data Bus (immediate);
MBR → IR	MBR → ABUS (immediate),
	ABUS → IR (delayed);
MBR → AC	MBR → MBUS (immediate),
	MBUS → ALU B (immediate),
	ALU PASS B (immediate),
	ALU Result → RBUS (immediate),
	RBUS → AC (delayed);

Finite State Machines for Simple CPUs

Mapping onto Datapath Operations

Relationship between register transfer and microoperations:

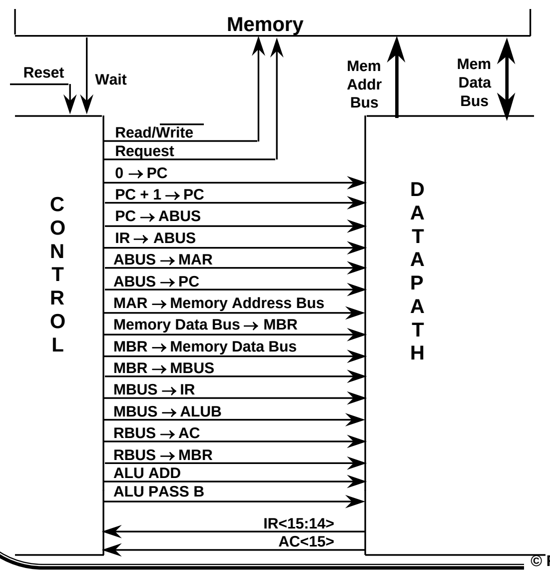
Register Transfer	Microoperations
AC → MBR	AC → RBUS (immediate),
	RBUS → MBR (delayed);
AC + MBR → AC	AC → ALU A (immediate),
	MBR → MBUS (immediate),
	MBUS → ALU B (immediate),
	ALU ADD (immediate),
	ALU Result → RBUS (immediate),
	RBUS → AC (delayed);
IR<13:0> → MAR	IR → ABUS (immediate),
	ABUS → IR (delayed);
IR<13:0> → PC	IR → ABUS (immediate),
	ABUS → PC (delayed);
1 → Read/Write	Read (immediate);
0 → Read/Write	Write (immediate);
1 → Request	Request (immediate);

Special microoperations for AC → ALU and ALU Result → RBUS not strictly necessary since these connections can be hardwired

Finite State Machines for Simple CPUs

Mapping onto Datapath Operations

Revised microoperation signal flow



5 inputs

make sure that Reset and Wait are synchronized

16 datapath control lines

2 memory control lines

Controller Implementation

Chapter Summary

- ***Basic organization of the Von Neumann computer***

  Separation of processor and memory

- ***Datapath connectivity***

- ***Control Unit Organization***

  Register transfer operation