# CPE300: Digital System Architecture and Design

Fall 2011

MW 17:30-18:45 CBC C316

Layered View of the Computer

http://www.egr.unlv.edu/~b1morris/cpe300/

# Outline

- Recap
- Assembly/Machine Programmer View
- Computer Architect View
- Digital Logic Designer View

# The General Purpose Machine

- Computers are more than just the personal computers we commonly use

- All around and in everything
- Different varieties for given applications
  - Supercomputers, automobiles, thermostats, toys, etc.

# 5 Classical Computer Components

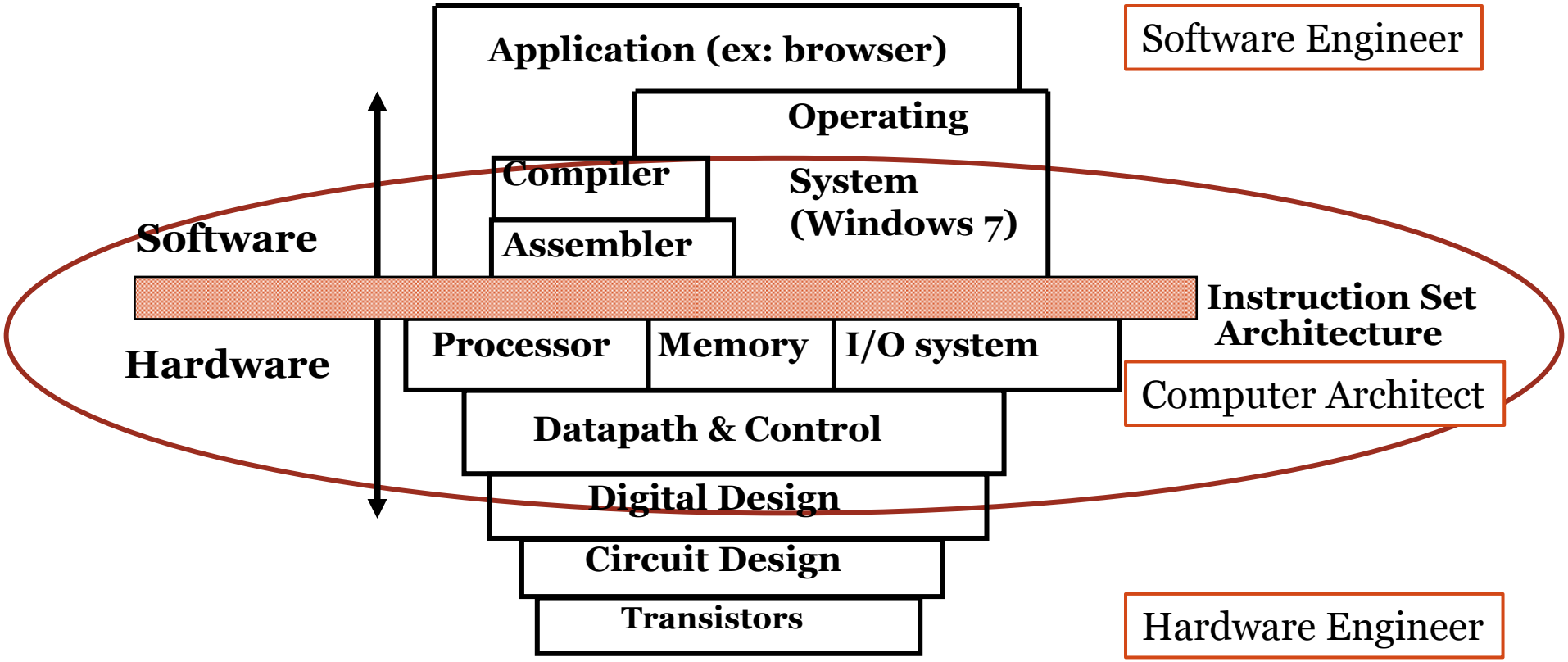
- Control – the "brain"
- Datapath – the "brawn"

} Processor

- Memory – where programs and data live when running
- Input
  - E.g. keyboard, mouse, disk
- Output
  - E.g. disk, display, printer

} Devices

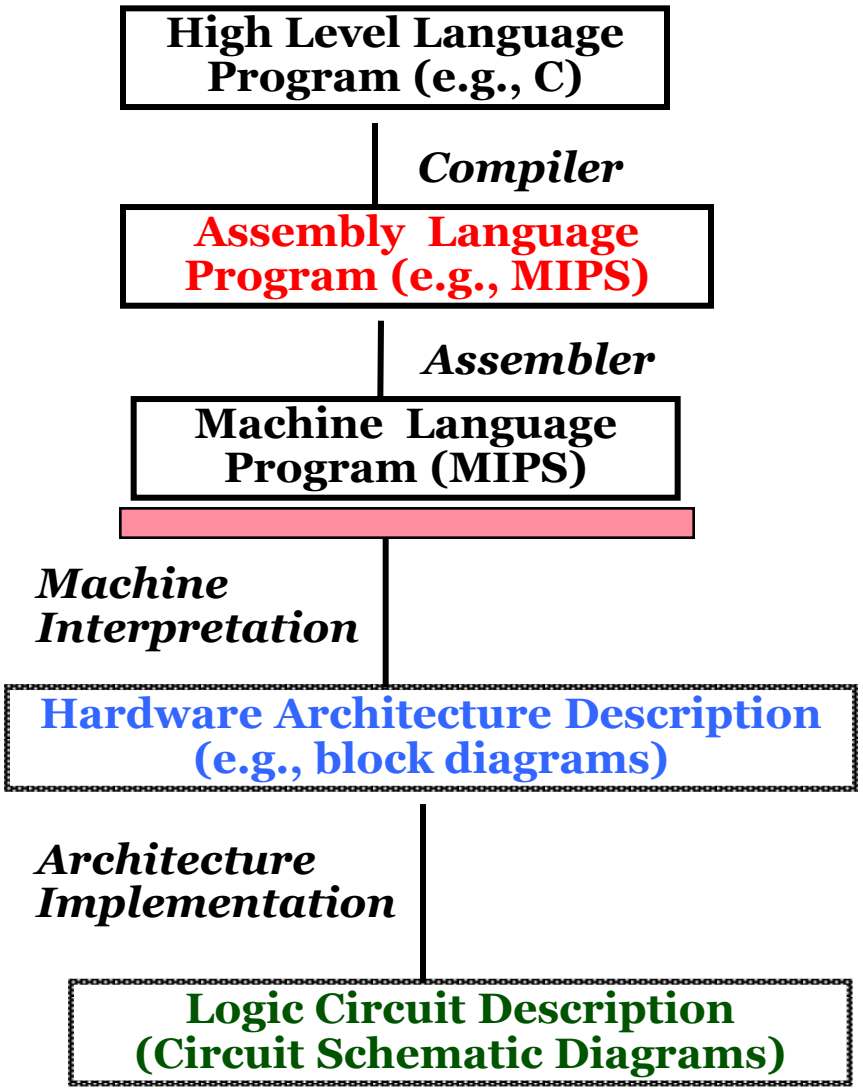
# Machine Structures




- Coordination of many levels of abstraction

Slide from UC Berkeley CS61C

# Levels of Representation/Interpretation

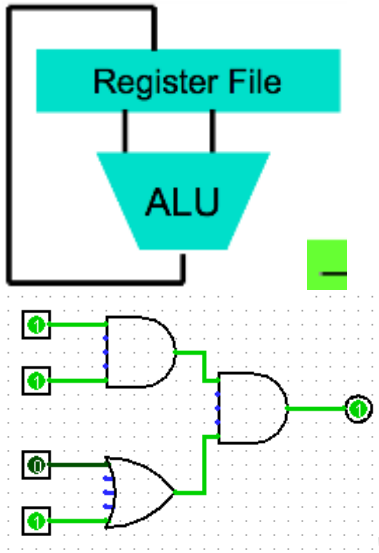



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

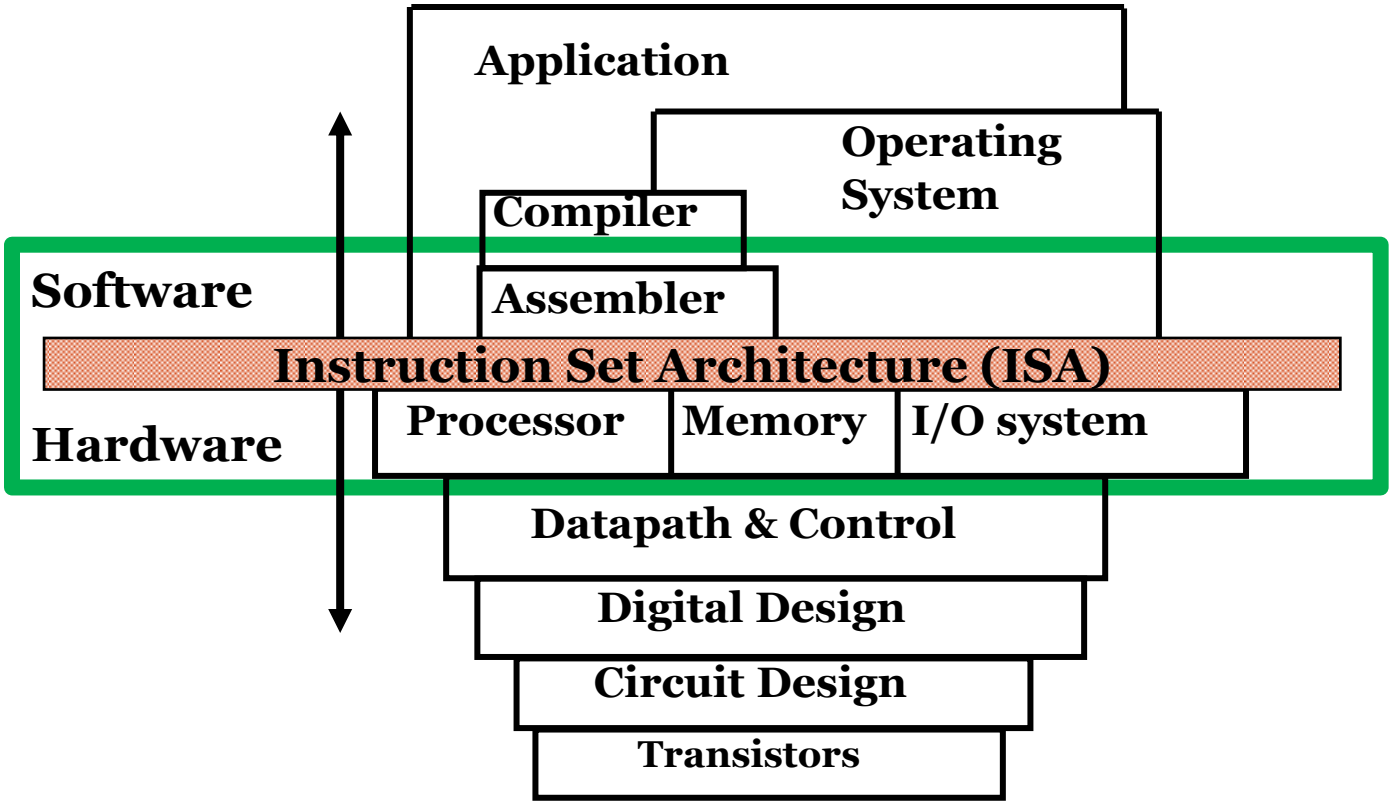



Slide from UC Berkeley CS61C

# Three Important Views of Computer

- Assembly/Machine Language Programmer
  - Concerned with behavior and performance of machine when programmed at lowest level (machine language)
- Computer Architect
  - Concerned with design and performance at (sub) system levels
- Logic Designer
  - Concerned with design at the digital logic level

# Assembly/Machine Layer




Assembly/Machine Language Programmer View

# Machine/Assembly Programmer's View

- Machine language
  - Set of fundamental instructions the machine can execute
  - Expressed as patterns of 0's and 1's
- Assembly language
  - Alpha numeric equivalent of machine language
  - Human oriented mnemonics (human readable)

# Machine and Assembly Language

- Assembler:
  - Computer program that transliterates (one-to-one mapping) assembly to machine language
  - Computer's native language is assembly/machine language

| MC68000 Assembly Language | Machine Language |
|---|---|
| MOVE.W D4, D5 | 0011 101 000 000 100 |
| ADDI.W #9, D2 | 00000001 10 111 100<br>0000 0000 0000 1001 |

**Table 1.2 Two Motorola MC68000 instructions**

# Instruction (Dis)Assembly

- How to convert between assembly and machine languages?
  - See “Programmers Reference Manual”
- Specify instruction fields
  - Opcode – operation code (what to do)
  - Operands – elements opcode will operate on
  - Each instruction has a different definition but there is structure
- Assembly/machine code is unique to the particular machine program runs on
  - Cannot port between machines like HLL (C) code

# Assembly Example

| MC68000 Assembly Language | Machine Language |
|---|---|
| MOVE.W D4,D5 | 0011 101 000 000 100 |

- Reference manual excerpt

**MOVE** **Move Data from Source to Destination** **MOVE**
**(M68000 Family)**

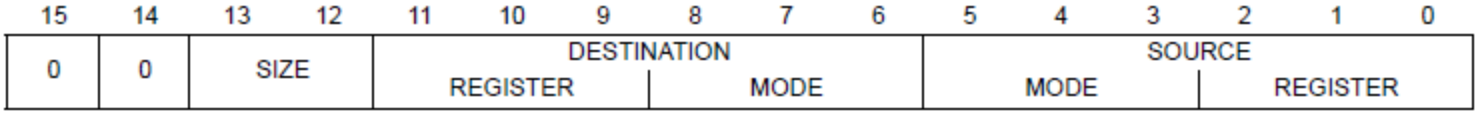
**Operation:** Source → Destination

**Assembler Syntax:** MOVE < ea > , < ea >

**Attributes:** Size = (Byte, Word, Long)

**Description:** Moves the data at the source to the destination location and sets the condition codes according to the data. The size of the operation may be specified as byte, word, or long. Condition Codes:

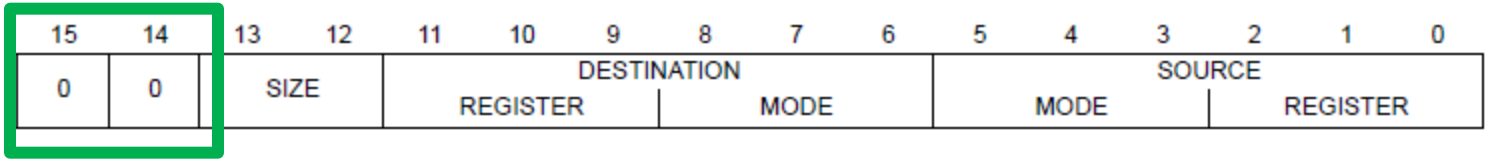
**Instruction Format:**



# Assembly Example - Opcode

| MC68000 Assembly Language | Machine Language |
|---|---|
| MOVE.W D4,D5 | 00 11 101 000 000 100 |

**Instruction Format:**

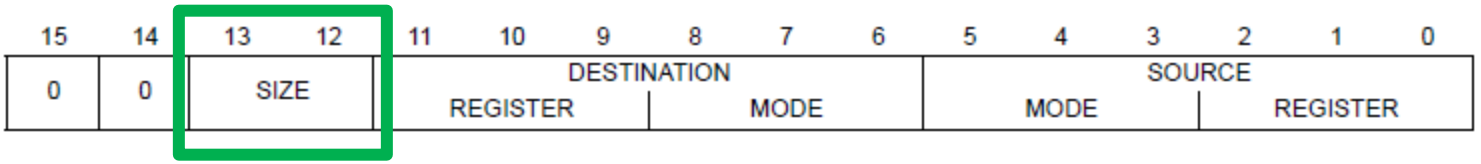



- Opcode for MOVE
- 2 MSB (bits [15, 14]) default to 0

# Assembly Example - Opcode

| MC68000 Assembly Language | Machine Language |
|---|---|
| MOVE.W D4,D5 | 0011 101 000 000 100 |

**Instruction Format:**




- Bits [13, 12] indicate data size

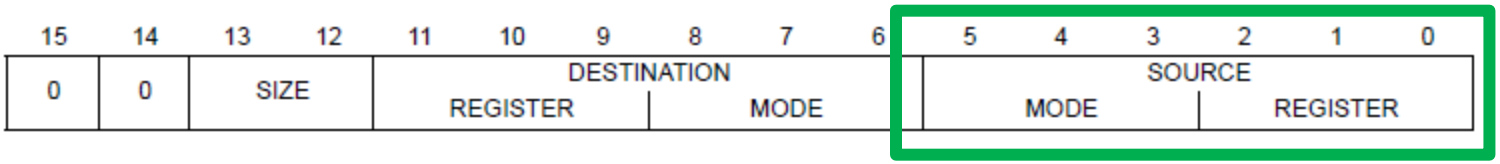
Size field—Specifies the size of the operand to be moved.
01 — Byte operation
11 — Word operation
10 — Long operation

  - Move.W for a word → 11

# Assembly Example – Source Operand

| MC68000 Assembly Language | Machine Language |
|---|---|
| MOVE.W D4, D5 | 0011 101 000 000 100 |

**Instruction Format:**




- Source operand (bits [5, 0])

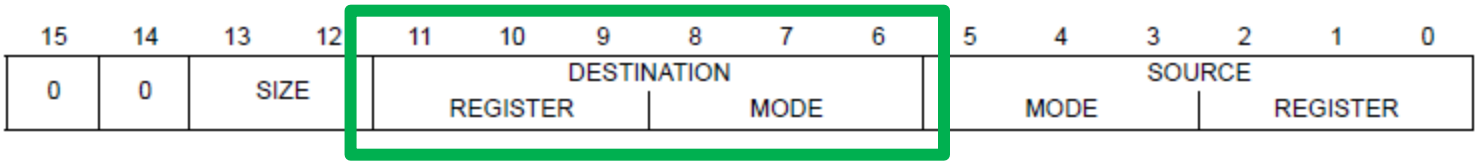
| Addressing Mode | Mode | Register |
|---|---|---|
| Dn | 000 | reg. number:Dn |

  - Mode = 000
  - Register number = 4 = 100B

# Assembly Example – Destination Operand

| MC68000 Assembly Language | Machine Language |
|---|---|
| MOVE.W D4, D5 | 0011 101 000 000 100 |

**Instruction Format:**




- Source operand (bits [5, 0])

| Addressing Mode | Mode | Register |
|---|---|---|
| Dn | 000 | reg. number:Dn |

  - Mode = 000
  - Register number = 5 = 101B

# Stored Program Concept

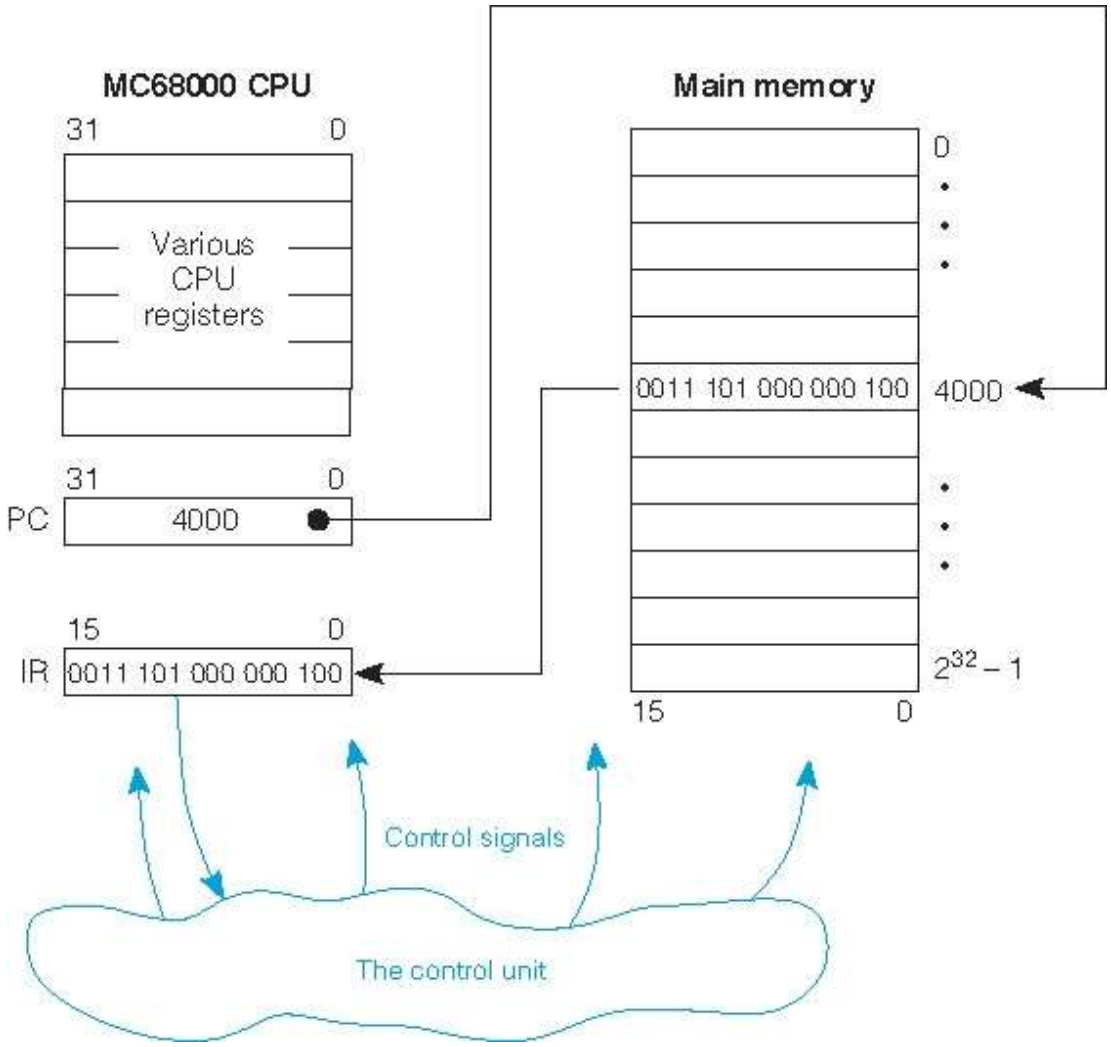
- Big idea – Everything is data!

***The stored program concept says that the program is stored with data in the computer's memory. The computer is able to manipulate it as data—for example, to load it from disk, move it in memory, and store it back on disk.***

  - Bits are just bits – up to computer to decide how to interpret
  - Basic operating principle of every computer

# Fetch-Execute Cycle

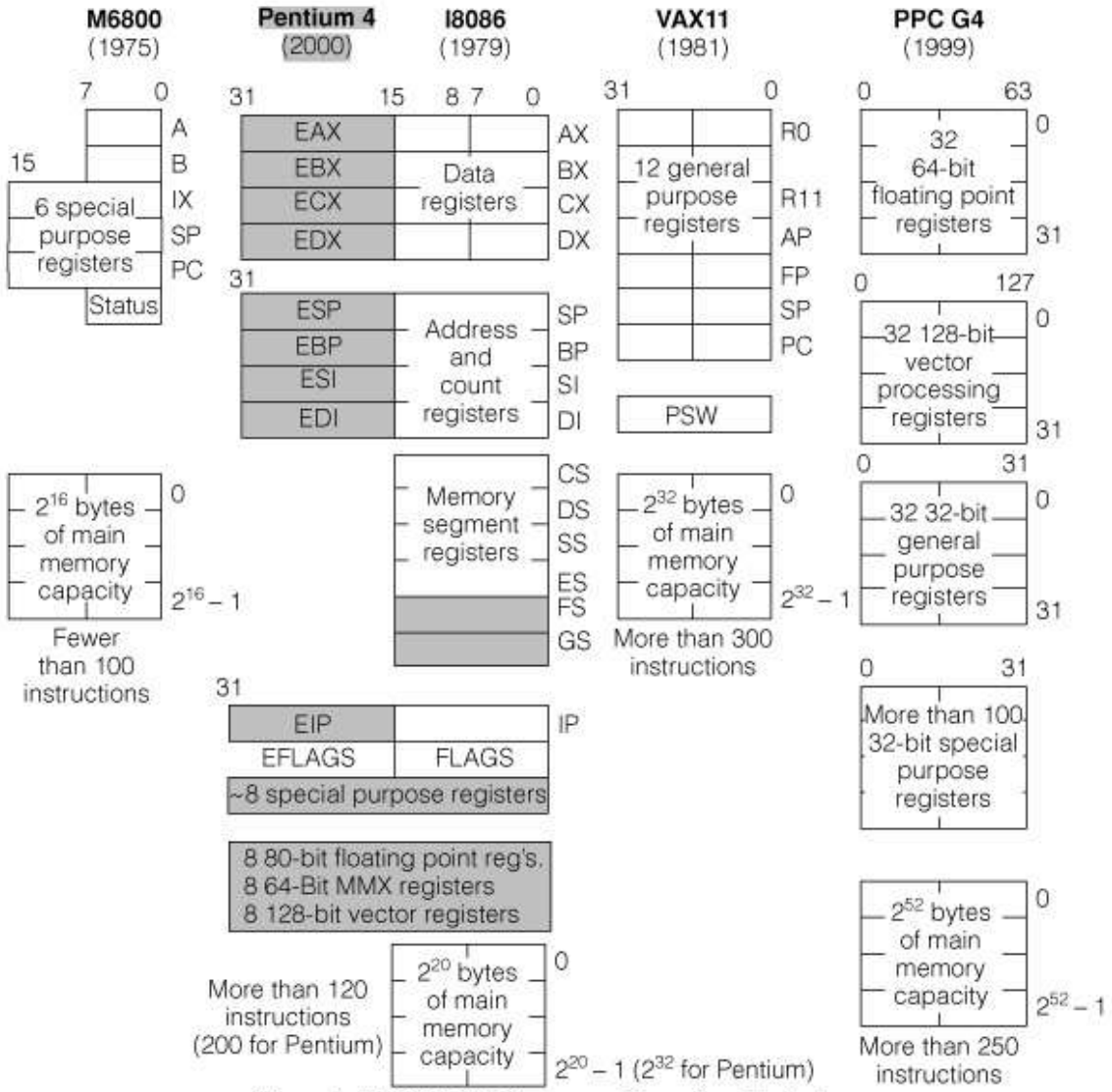
- Instruction fetched from memory
  - Stored in instruction register (IR)
- Instruction decoded through control unit and executed
- Next instruction is available in program counter (PC) register
  - PC must be incremented based on instruction size
  - 2 byte instructions here so PC incremented by 2



# Instruction Set Architecture (ISA)

- Instruction set: the collection of all machine operations.
- Programmer sees set of instructions, along with the machine resources manipulated by them.
- ISA includes
  - instruction set,
  - memory, and
  - programmer accessible registers of the system.
- There may be temporary or scratch-pad memory used to implement some function is not part of ISA.
  - “Non Programmer Accessible.”

# Programmer Models of 4 Commercial Machines



# RISC vs. CISC Machines

- Complex instruction set computers (CISC)
  - Primary goal to minimize number of assembly line for a task
    - Processor understands many operations
  - Operates directly on computer memory banks
  - Assembler compiler has little work for translation from HLL
  - Shorter code requires less memory
  - Hardware emphasis
- Reduced instruction set computers (RISC)
  - Use simple instructions that complete within a clock cycle
  - Typically longer code (e.g. load, operation, store)
  - Approximately same speed as CISC
  - Require less hardware (transistors) to implement
    - Use space for more registers
  - Pipelining is possible
  - Software emphasis

# Machine, Processor, and Memory State

- The Machine State: contents of all registers in system, accessible to programmer or not
- The Processor State: registers internal to the CPU
- The Memory State: contents of registers in the memory system
- Maintaining or restoring the machine and processor state is important to many operations, especially procedure calls and interrupts

# Data Type: HLL vs. Machine Language

- HLL's provide type checking
  - Verifies proper use of variables at compile time
  - Allows compiler to determine memory requirements
  - Helps detect bad programming practices
- Most machines have no type checking
  - The machine sees only strings of bits
  - Instructions interpret the strings as a type: usually limited to signed or unsigned integers and FP #s
  - A given 32 bit word might be an instruction, an integer, a FP #, or four ASCII characters
- (Data is data!)

# Instruction Classes

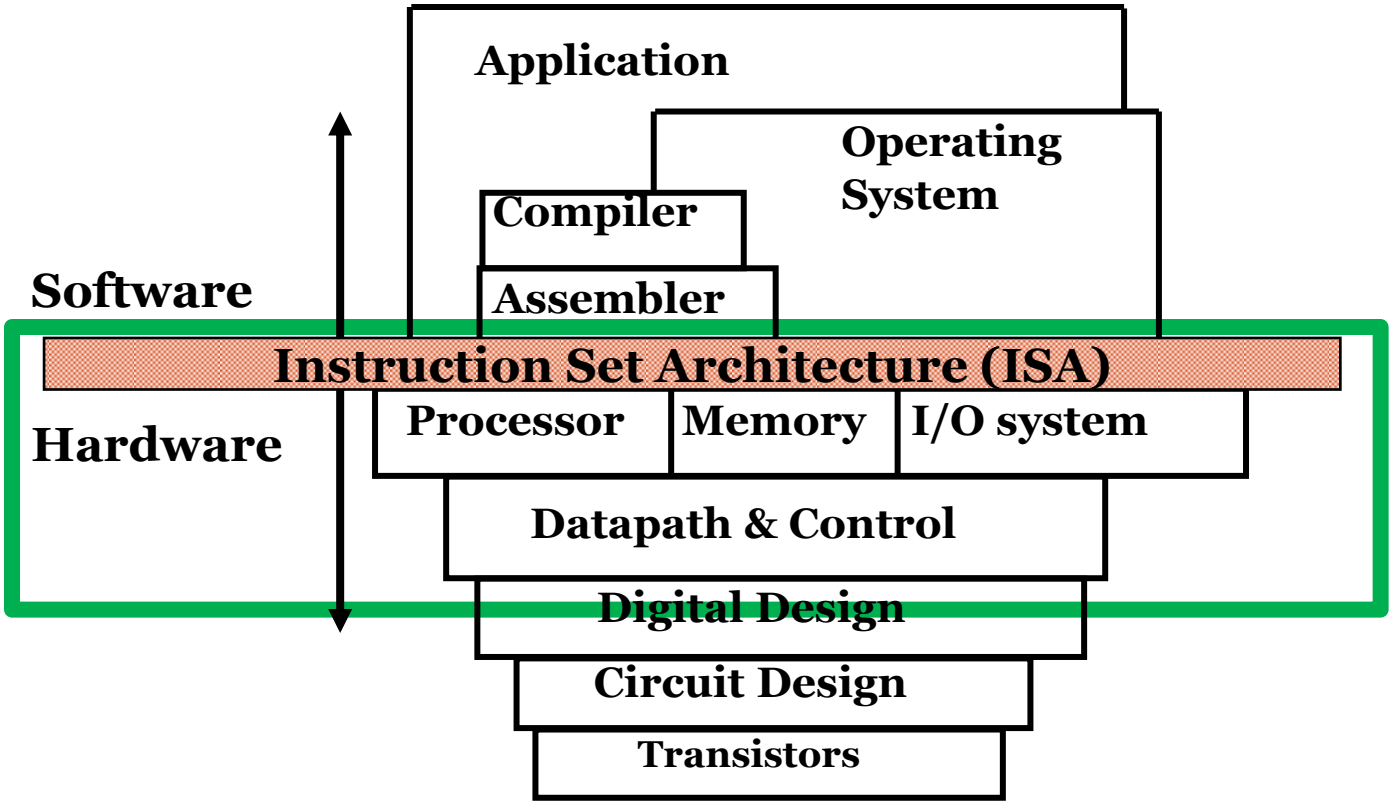
- 3 classes of machine instructions
  - Data movement
  - Arithmetic and logic
  - Control flow
- The compiler writer must develop mapping for each language-machine pair

| Instruction Class | C | VAX Assembly Language |
|---|---|---|
| Data Movement | a = b | MOV b, a |
| Arithmetic/logic | b = c + d*e | MPY d, e, b<br>ADD c, b, b |
| Control flow | goto LBL | BR LBL |

# Users of Assembly Language

- The machine designer
  - must implement and trade-off instruction functionality
- The compiler writer
  - must generate machine language from a HLL
- The writer of time or space critical code
  - Performance goals may force program specific optimizations of the assembly language
- Special purpose or imbedded processor programmers
  - Special functions and heavy dependence on unique I/O devices can make HLL's useless

# Computer Architect Layer



# Computer Architect View

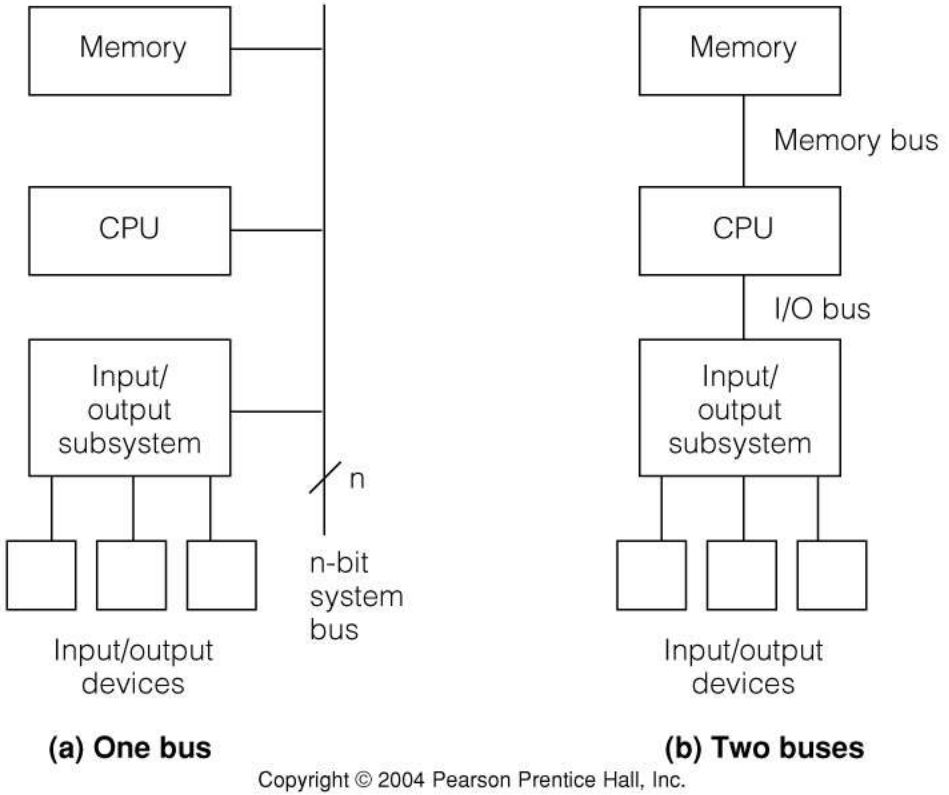
- Architect is concerned with design & performance
- Designs the ISA for optimum programming utility and optimum performance of implementation
- Designs the hardware for best implementation of the instructions
  - CPU, memory, peripheral devices
- Uses performance measurement tools, such as benchmark programs, to see that goals are met
- Balances performance of building blocks such as CPU, memory, I/O devices, and interconnections
- Meets performance goals at lowest cost

# Shared Interconnections Via Buses

- Interconnections are very important to computer
- Most connections are shared
- A bus is a time-shared connection or multiplexer
- A bus provides a data path and control
- Buses may be serial, parallel, or a combination
  - Serial buses transmit one bit at a time
  - Parallel buses transmit many bits simultaneously on many wires

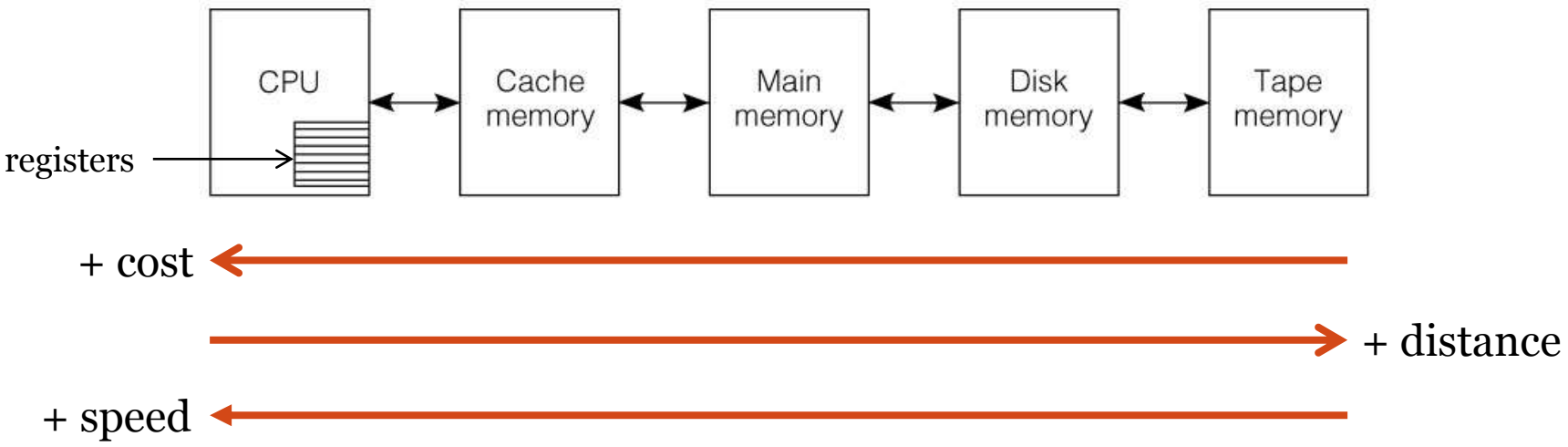
# Example 1 or 2 Bus Architectures

- (a) Single shared bus of n lines
  - Simple connection
  - Only one system active at a time
- (b) 2-bus system
  - Separates/isolates memory and I/O
  - Activity can be present in both buses concurrently
    - Speedup because I/O is typically slow

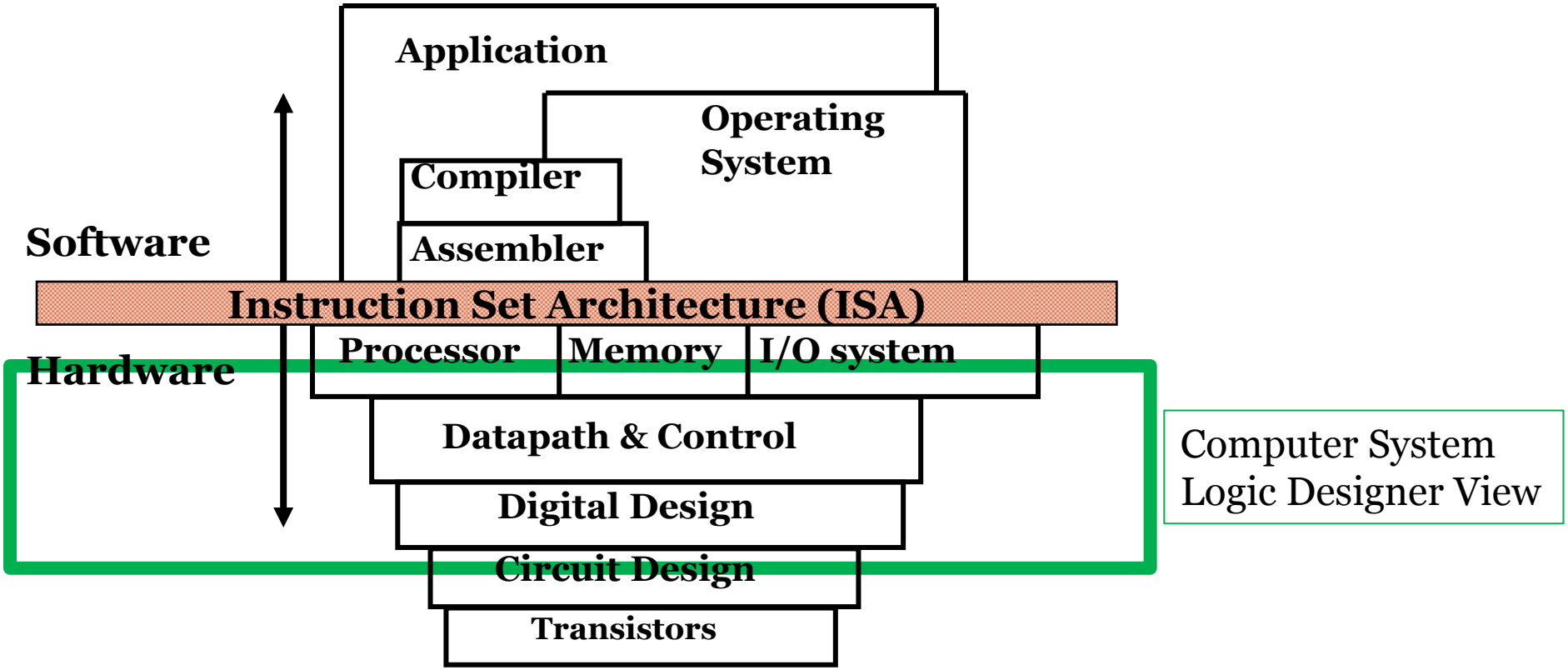


# Memory System

- Modern computers have a hierarchy of memories
  - Allow tradeoffs of speed/cost/volatility/size, etc



# Logic Designer Layer



# Logic Designer View

- Designs the machine at the logic gate level
- The design determines whether the architect meets cost and performance goals
- Architect and logic designer can often be the same person/team

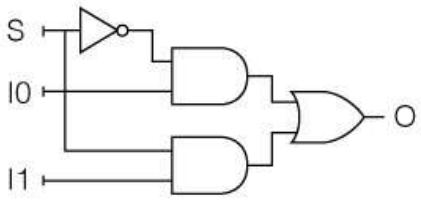
# Implementation Domains

- An implementation domain is the collection of devices, logic levels, etc. which the designer uses.

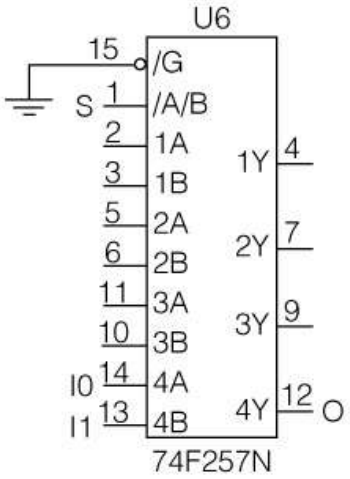
- Domain is usually abstracted
- Possible implementation domains
  - VLSI on silicon
  - TTL or ECL chips
  - Gallium Arsenide chips
  - PLA's or sea-of-gates arrays
  - Fluidic logic or optical switches

# Implementation Domain Examples

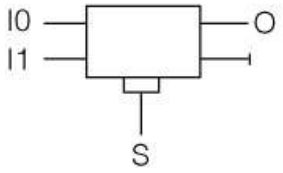
- 2 to 1 multiplexer in three different implementation domains
  - Generic logic gates (abstract domain)
  - National Semiconductor FAST Advanced Schottky TTL (vlsi on Si)
  - Fiber optic directional coupler switch (optical signals in $LiNbO_3$)




(a) Abstract view of Boolean logic




(b) TTL implementation domain




(c) Optical switch implementation

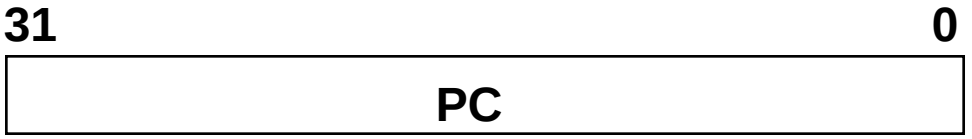
# Classical vs. Computer Logic Design

- Computer design is complex
  - Traditional FSM techniques can be used “in the small”
- There is a natural separation between data and control
  - Data path: storage cells, arithmetic, and their connections
  - Control path: logic that manages data path information flow
- Well defined “building” blocks are used repeatedly
  - Multiplexers, decoders, adders, etc

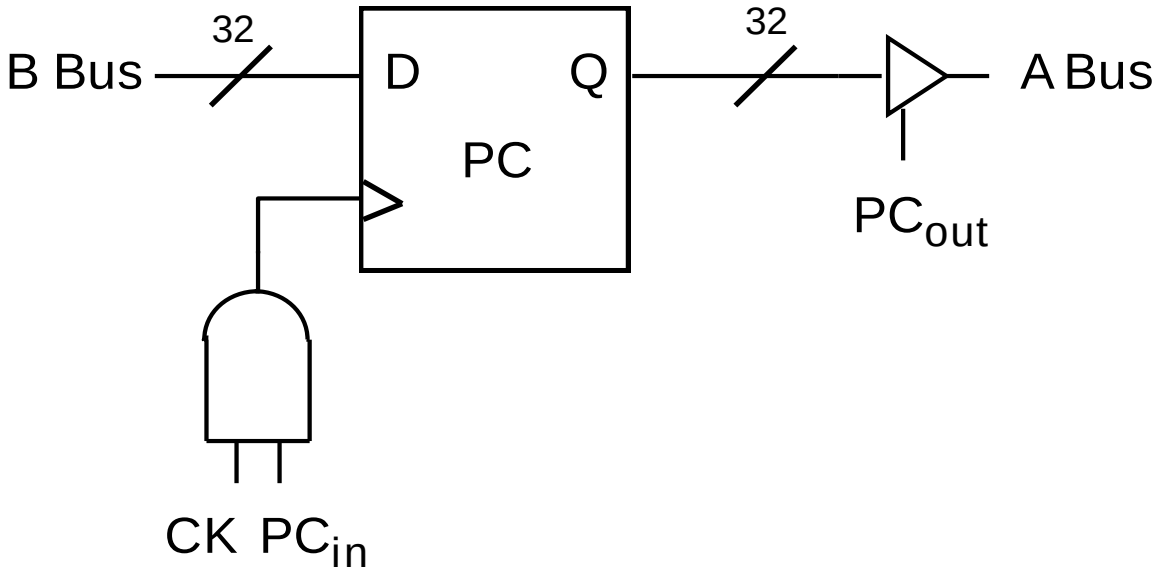
# Logic Designer CPU

- Cares about registers, ALU, and buses
- Most importantly, concerned with control via PC

**Programmer:**




**Logic Designer (Fig 1.8):**



# Concluding Remarks

- 3 different views of machine structure and function
- Machine/assembly language view: registers, memory cells, instructions.
  - PC, IR,
  - Fetch-execute cycle
  - Programs can be manipulated as data
  - No, or almost no data typing at machine level
- Architect views the entire system
  - Concerned with price/performance, system balance
- Logic designer sees system as collection of functional logic blocks.
  - Must consider implementation domain
  - Tradeoffs: speed, power, gate fanin, fanout