CPE300: Digital System Architecture and Design

Fall 2011

MW 17:30-18:45 CBC C316

Instruction Set Architecture

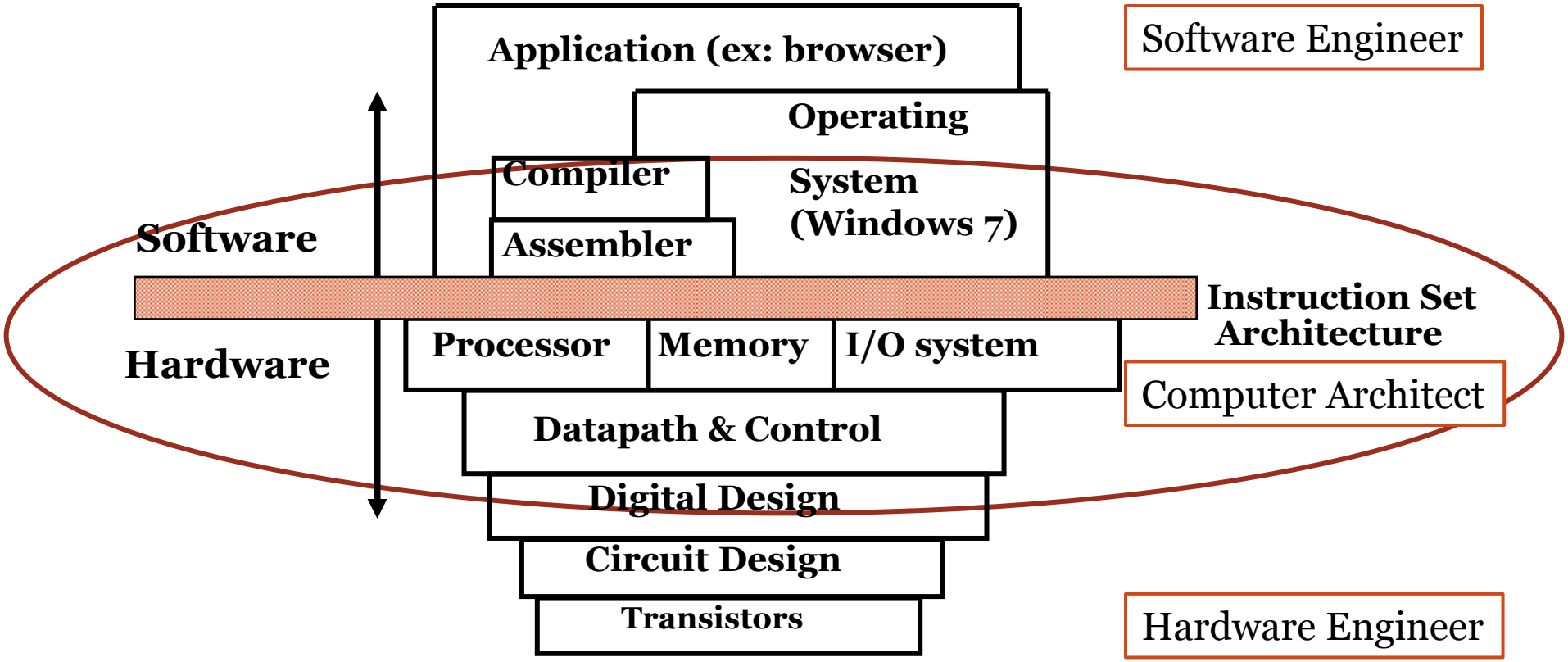
09072011

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

Outline

- Recap
- Instruction Sets
- Registers
- x-Address Machines
- Addressing Modes

Machine Structures



- Coordination of many levels of abstraction

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

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

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.”

Chapter 2: Machines, Machine Languages, and Digital Logic

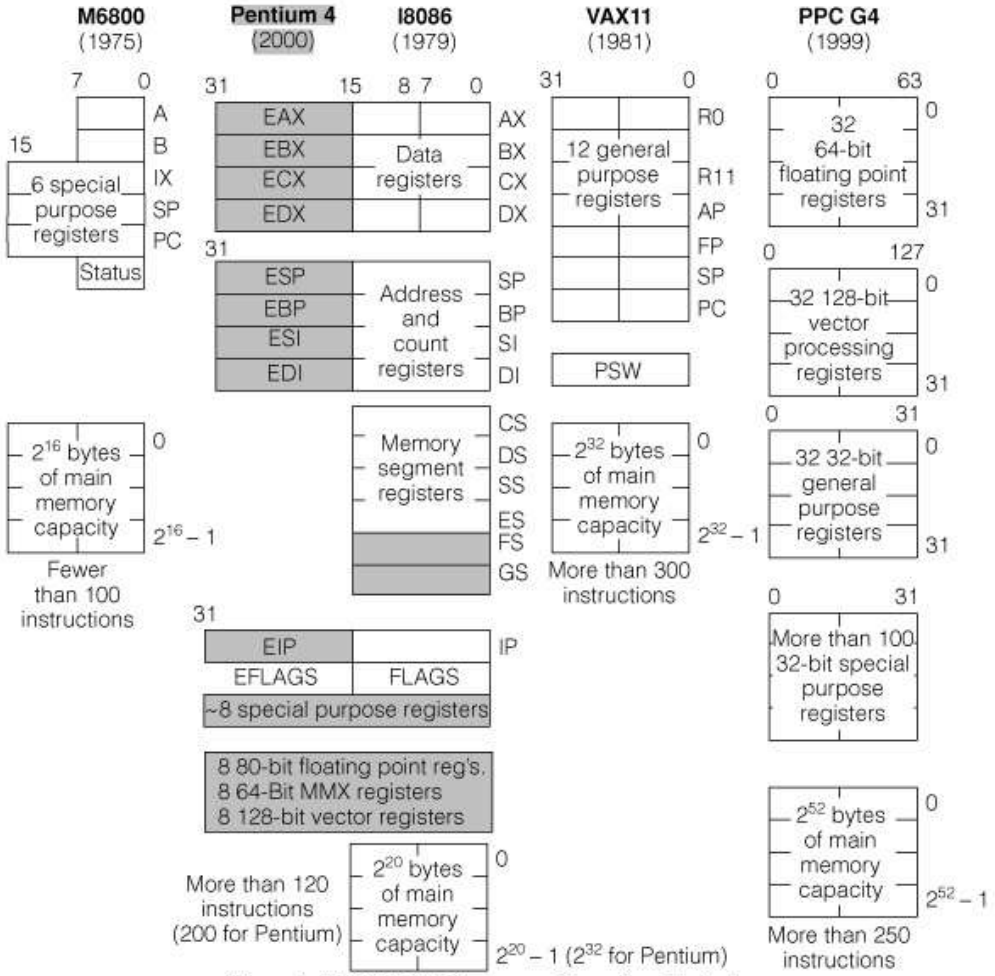
- Instruction sets
- Simple RISC Computer (SRC)
- Register Transfer Notation (RTN)
- Mapping of register transfers to digital logic circuits

ISA Components

- Sometimes known as *The Programmers Model* of the machine
- Storage cells
  - General and special purpose registers in the CPU
  - Many general purpose cells of same size in memory
  - Storage associated with I/O devices
- The Machine Instruction Set
  - The instruction set is the entire repertoire of machine operations
  - Makes use of storage cells, formats, and results of the fetch/execute cycle
  - i. e. Register Transfers
- The Instruction Format
  - Size and meaning of fields within the instruction
- The nature of the Fetch/Execute cycle
  - Things that are done before the operation code is known

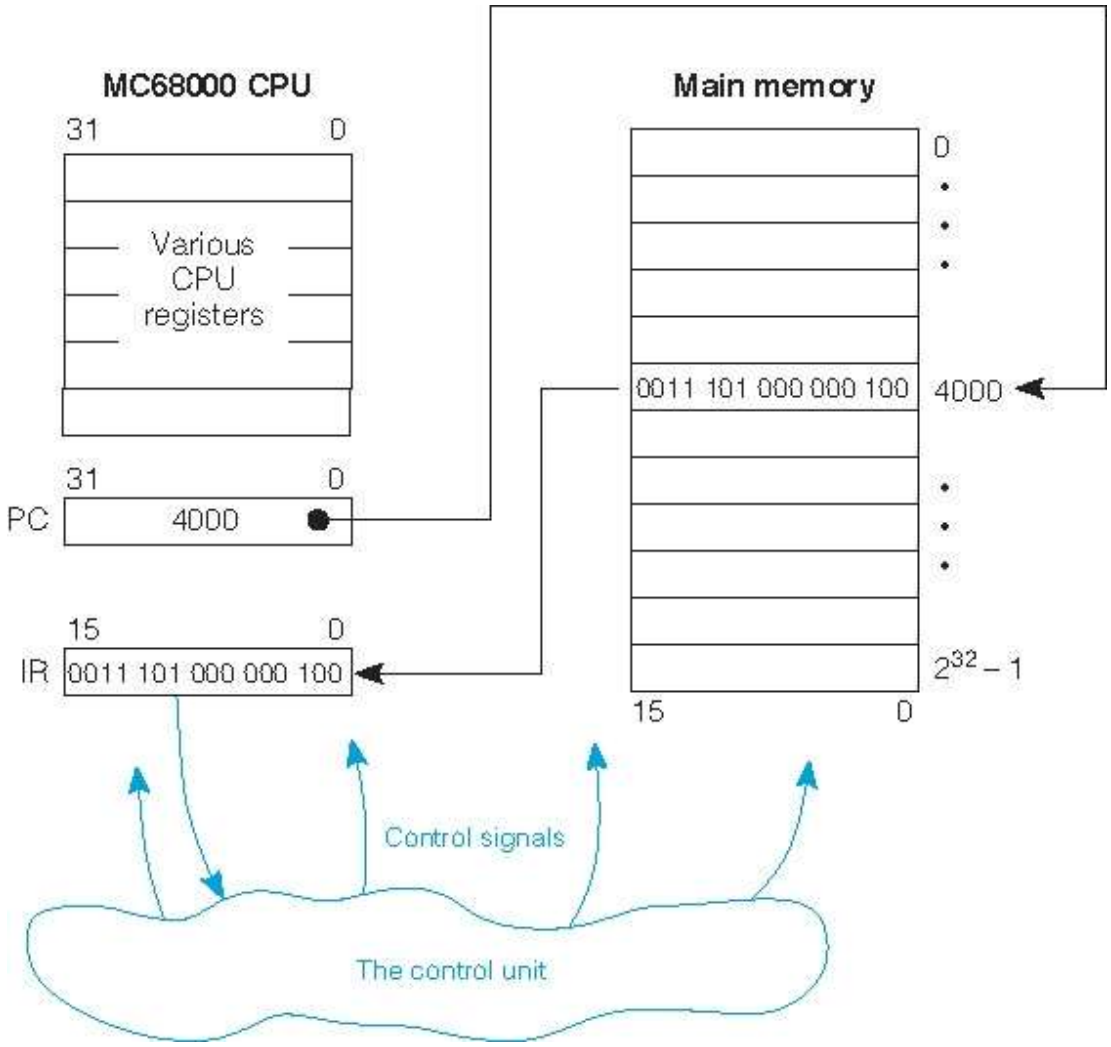
Fig. 2.1 Programmer's Models of Various Machines

- Various numbers and types of storage cells



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 Specification

- Which operation to perform
  - Op code: add, load, branch, etc.

```
add r0, r1, r3
```

- Where to find the operands
  - In CPU registers, memory cells, I/O locations, or part of instruction

```
add r0, r1, r3
```

- Place to store result
  - Again CPU register or memory cell

```
add r0, r1, r3
```

- Location of next instruction
  - The default is usually memory cell pointed to by program counter (PC)

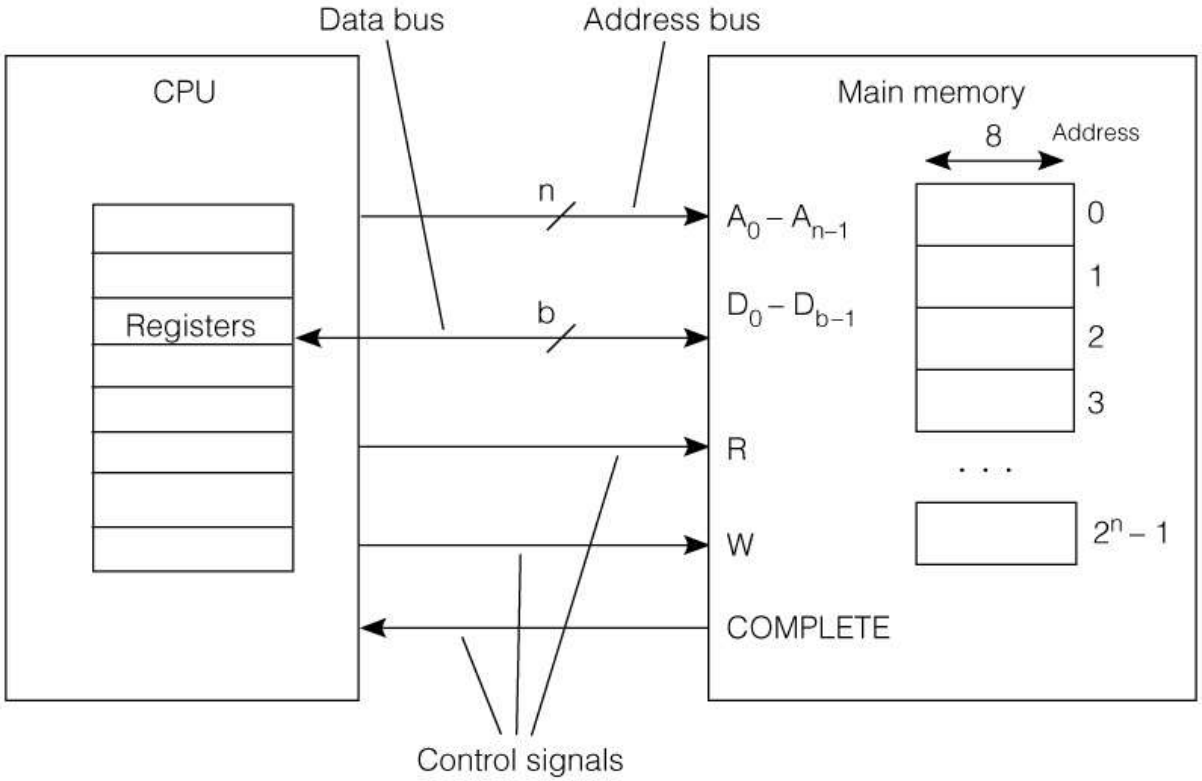
```
br endloop
```

3 Classes of Instructions

- Data movement instructions
  - Move data from a memory location or register to another memory location or register without changing its form
  - Load - source is memory and destination is register
  - Store - source is register and destination is memory
- Arithmetic and logic (ALU) instructions
  - Changes the form of one or more operands to produce a result stored in another location
  - Add, Sub, Shift, etc.
- Branch instructions (control flow instructions)
  - Any instruction that alters the normal flow of control from executing the next instruction in sequence
  - Br Loc, Brz Loc2,—unconditional or conditional branches

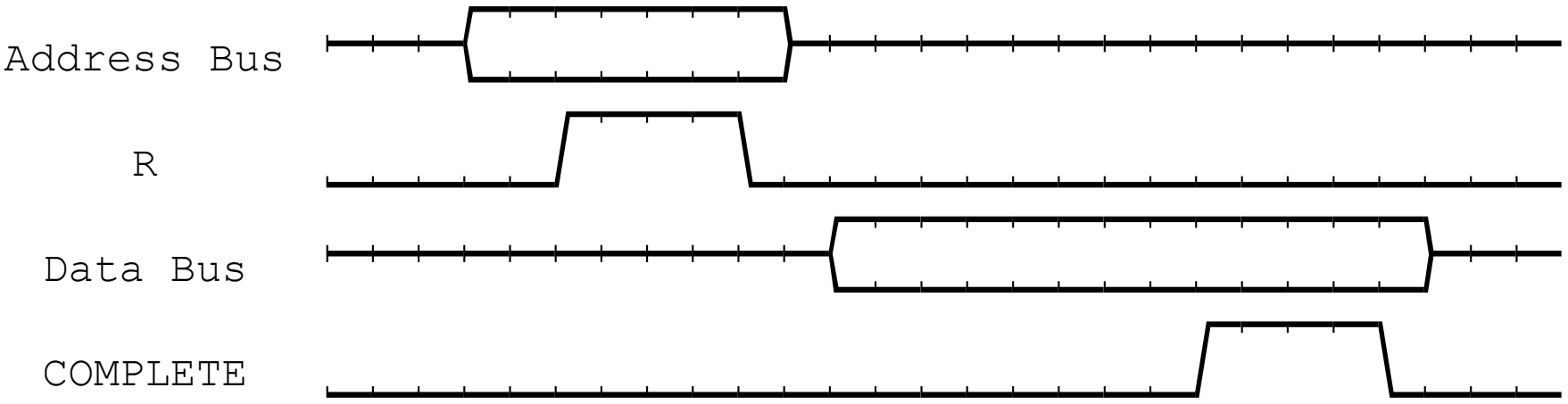
Memory Access: Read

- CPU applies desired address to Address lines `A0-An-1`
- CPU issues read command, `R`
- Memory returns the value at that address on Data lines `D0-Db-1` and asserts the `COMPLETE` signal



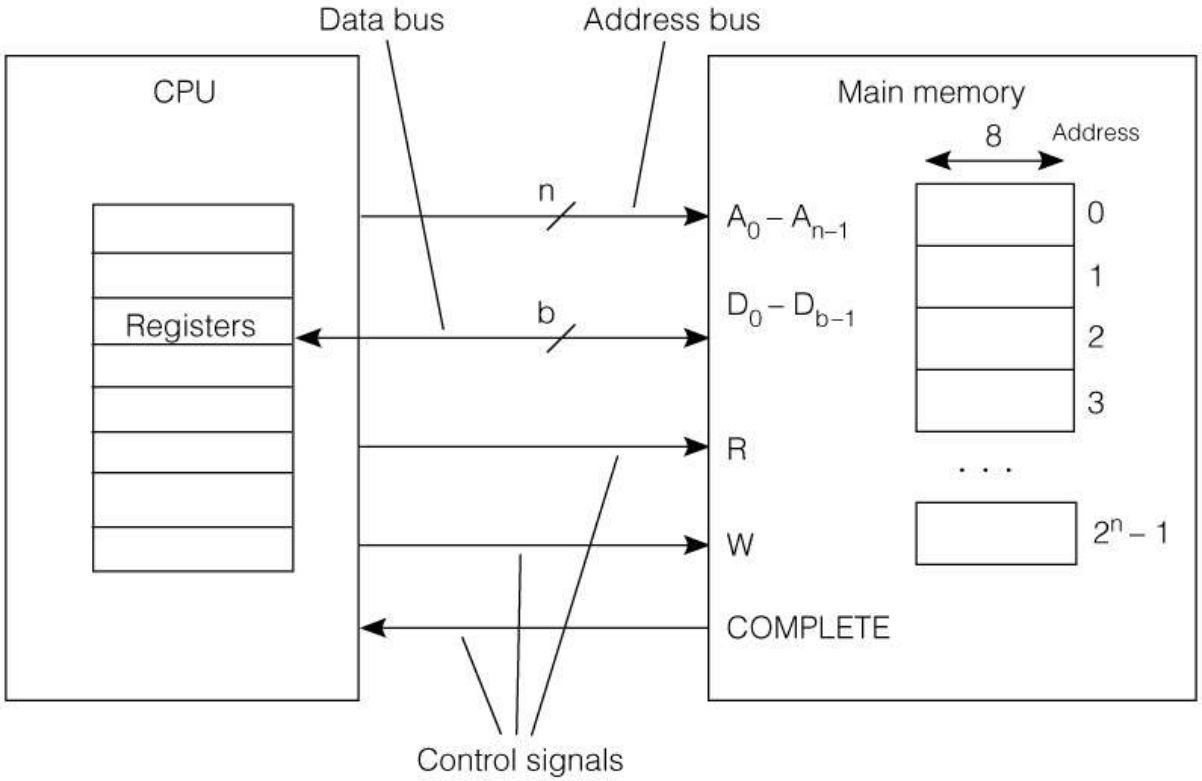
Read Timing

- Setup address
- Set R
- Data retrieved from memory
- Set COMPLETE



Memory Access: Write

- CPU applies desired address to Address lines `A0-An-1` and and data to be written on Data lines `D0-Db-1`
- CPU issues Write command, `W`
- Memory asserts the `COMPLETE` signal when the data has been written to memory



Data Movement Instructions

Instruction	Meaning	Machine
 `MOV A, B`	Move 16 bits from memory location A to B	VAX11
 `lwz R3, A`	Move 32 bits from memory location A to register 3	PPC601
 `li $3, 455`	Load 32 bit integer 455 into register 3	MIPS R3000
 `mov R4, dout`	Move 16 bits from register 4 to port dout	DEC PDP11
 `IN, AL, KDB`	Load byte from port KDB into accumulator	Intel Pentium
 `LEA.L (A0), A2`	Load address pointed at by A0 into A2	M68000

- Lots of variation, even with one instruction type
- Notice differences in direction of data flow left-to-right or right-to-left

ALU Instructions

Instruction	Meaning	Machine
`MULF A, B, C`	multiply the 32-bit floating point values at memory locations A and B, store at C	VAX11
`nabs r3, r1`	Store abs value of r1 in r3	PP0661
`ori $2, $1, 255`	Store logical OR (immediate) of reg $ 1 with 255 into reg $2	MIPS R3000
`DEC R2`	Decrement the 16-bit value stored in reg R2	DEC PDP11
`SHL AX, 4`	Shift the 16-bit value in reg AX left by 4 bits	Intel 8086

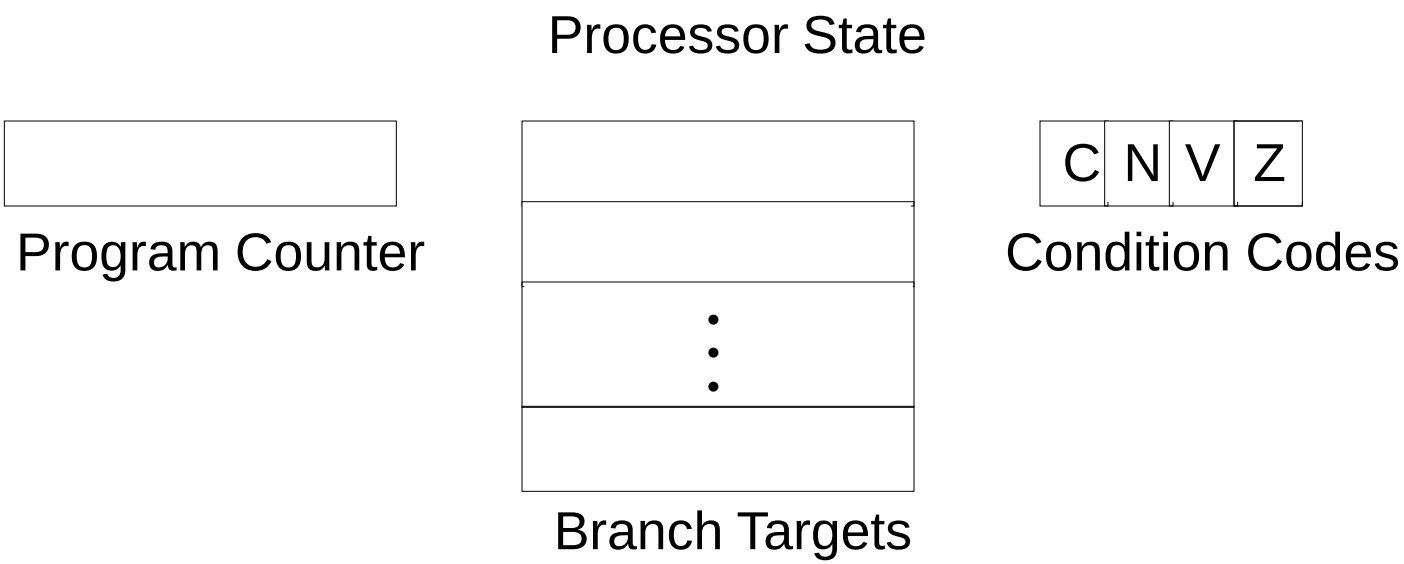
- Notice again the complete dissimilarity of both syntax and semantics
- RISC machines operate on registers, why?
  - Increased execution speed

Branch Instructions

Instruction	Meaning	Machine
`BLSS A, Tgt`	Branch to address Tgt if the least significant bit of memory location A is set (i.e. = 1)	VAX11
`bun r2`	Branch to location in R2 if result of previous floating point computation was NaN	PP0661
`beq $2, $1, 32`	Branch to location (PC + 4 + 32) if contents of $1 and $2 are equal	MIPS R3000
`SOB R4, Loop`	Decrement R4 and branch to Loop if R4 ≠ 0	DEC PDP11
`JCXZ Addr`	Jump to Addr if contents of register CX = 0	Intel 8086
`Eieio`	Enforce in-order execution of I/O	Power PC

Registers for Control

- Program counter usually contains the address of, or "points to" the next instruction
- Condition codes may control branch
- Branch targets may be contained in separate registers



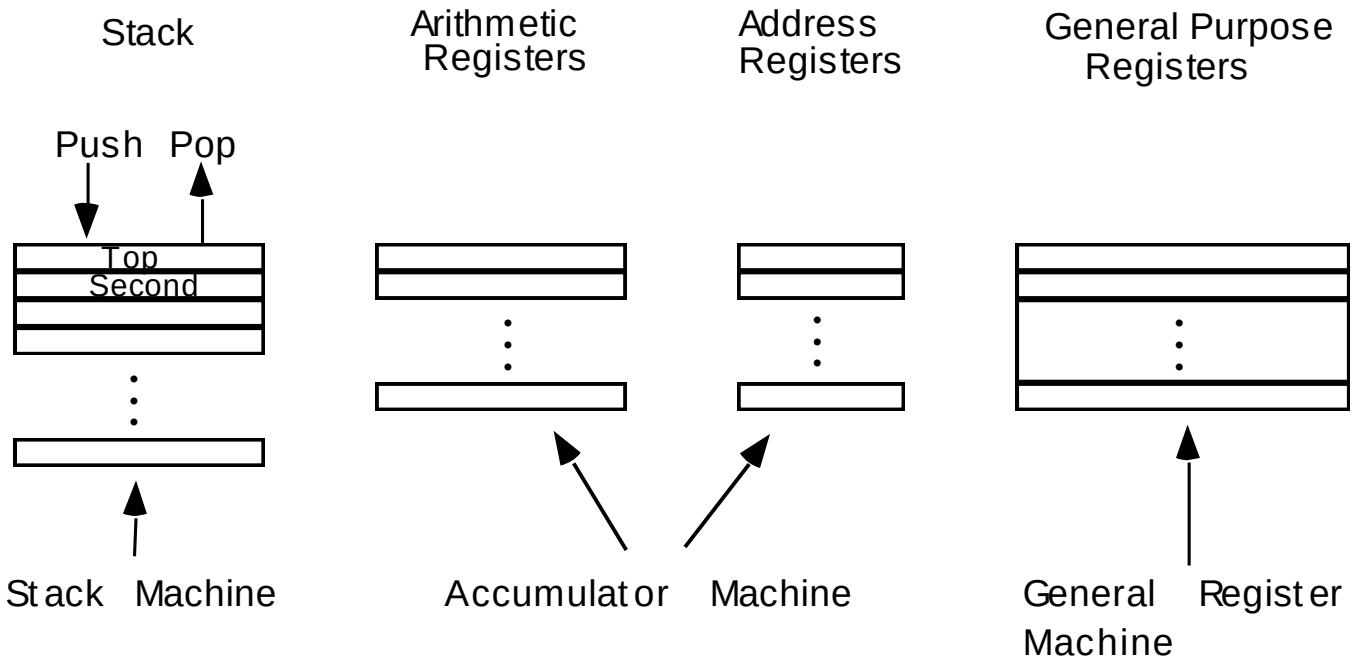
HLL Conditionals

- Typically no machine instruction mapping
  - Conditions computed by arithmetic instructions
  - Assembly conditional branch on result
- Program counter is changed to execute only instructions associated with true conditions

C Language	Assembly Language		
`if NUM==5`		`CMP.W #5, NUM`	;the comparison
`then SET=7`		`BNE L1`	;conditional branch
		`MOV.W #7, SET`	;action if true
	`L1`	…	;action if false

Register “Personality”

- Architecture classes are often based on
  - where the operands and result are located
  - how they are specified by the instruction.
  - They can be in CPU registers or main memory



Machine Instruction Encoding

- Instruction set must be converted into machine instructions
  - Bit patterns that specify instruction fields (e.g. opcode, operands, result, next instruction)
- Trade-off
  - Number of bits for specification
  - Size/flexibility of instructions
  - Also would like entire encoding to fit into a single word (RISC approach)

Hypothetical Machines

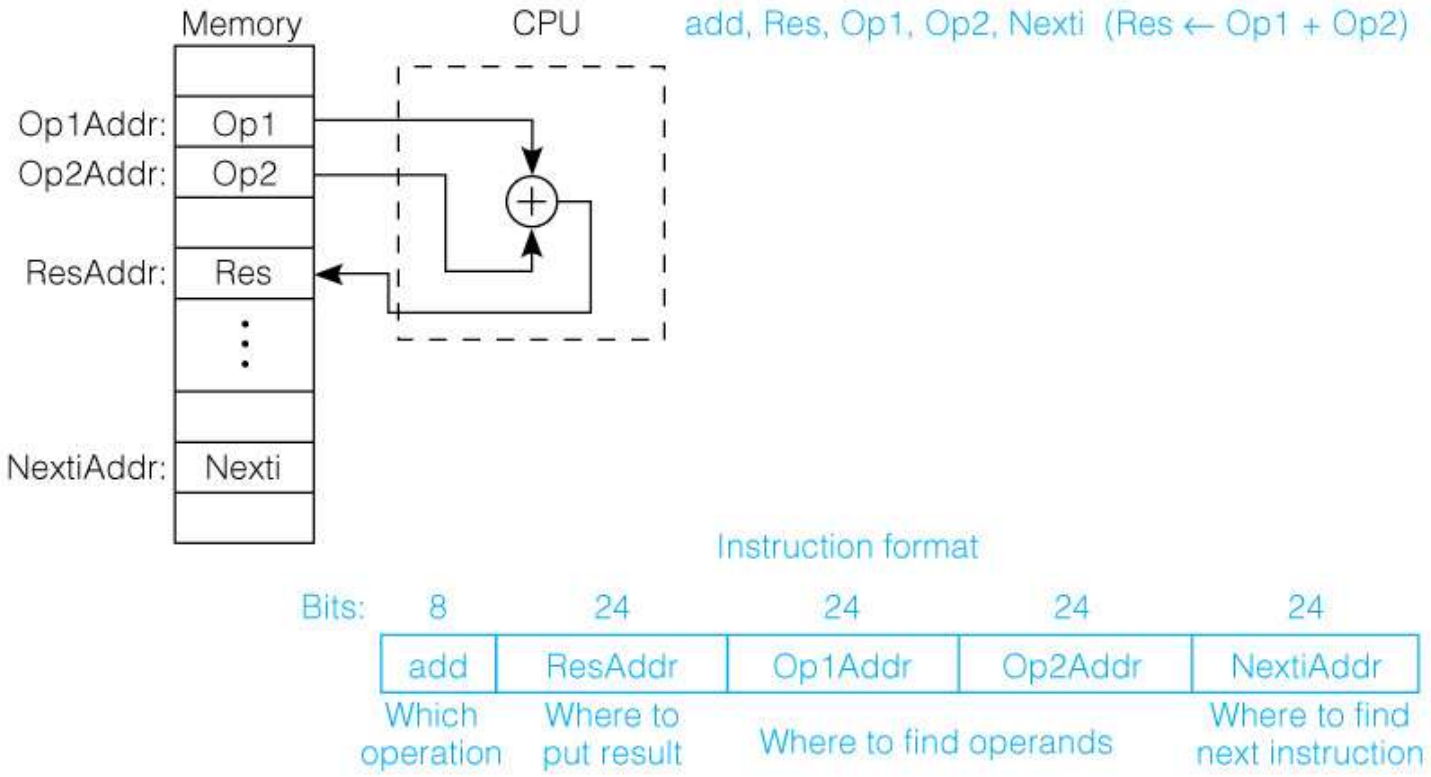
- Classify machine based on 2 operand (1 result) arithmetic (ALU) instruction
- 5 items to specify
  - Operation to perform
  - Location of first operand
  - Location of second operand
  - Location to store result
  - Location of next instruction to execute
- The key issue is “how many of these are specified by memory addresses, as opposed to being specified implicitly”

4,3,2,1,& 0 Address Instructions

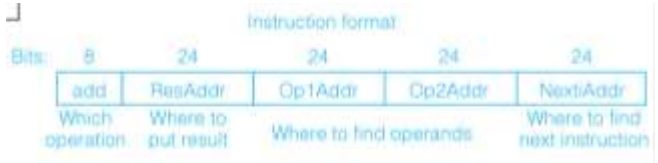
- 3 address instruction
  - Specifies memory addresses for both operands and the result
  - R ← Op1 op Op2
- 2 address instruction
  - Overwrites one operand in memory with the result
  - Op2 ← Op1 op Op2
- 1 address instruction
  - Single accumulator register to hold one operand & the result (no address needed)
  - Acc ← Acc op Op1
- 0 address
  - Uses a CPU register stack to hold both operands and the result
  - TOS ← TOS op SOS (TOS is Top Of Stack, SOS is Second On Stack)
- 4 address instruction
  - 3 address instruction + explicit definition of next address (rarely ever seen)

Fig. 2.3 The 4 Address Instruction

- Explicit addresses for operands, result, and next instruction
- Example assumes 24-bit addresses



4 Address Example



- Instructions in memory (typo in book)
  - Each address = 3 bytes (1 word)
  - Total bytes for ALU instruction = 4 x 3 + 1 = 13
  - Total words = 4 x 1 + 1 = 5
- Memory access
  - Instruction fetch = 5 words
  - Operands = 2 words (read)
  - Result = 1 word (write)
  - Address = 1 word (read)
  - Total = 5 + 2 + 1 = 8 (ignore address)
- Rarely used because of large instruction size and number of memory access makes

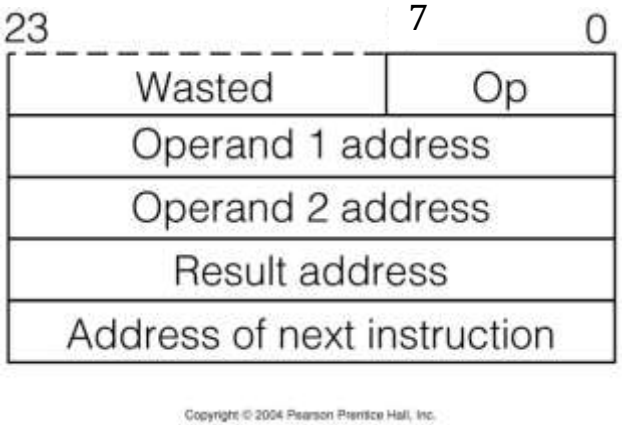
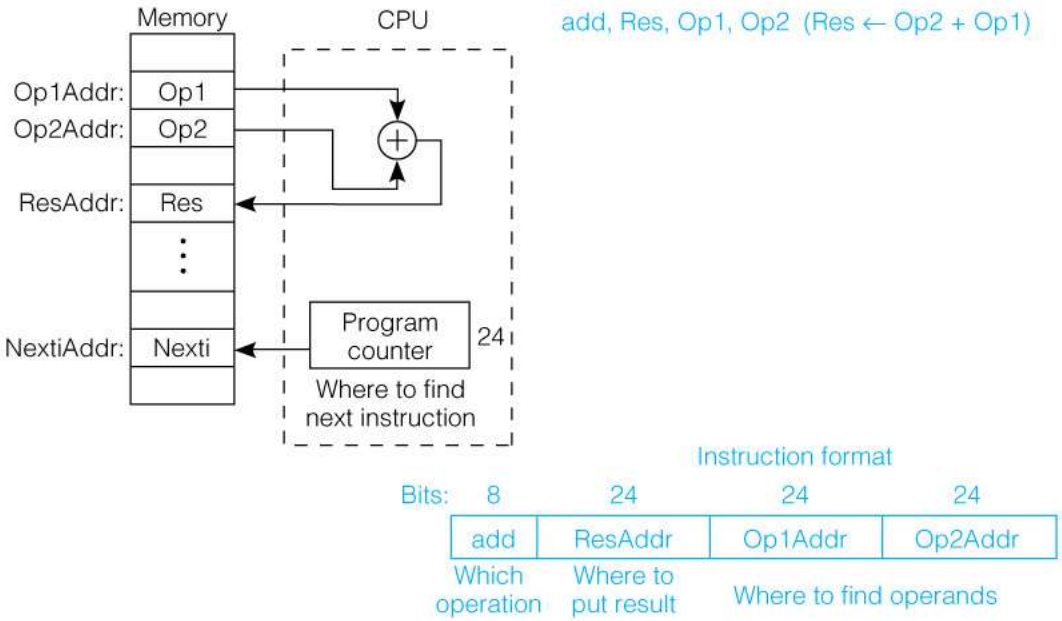
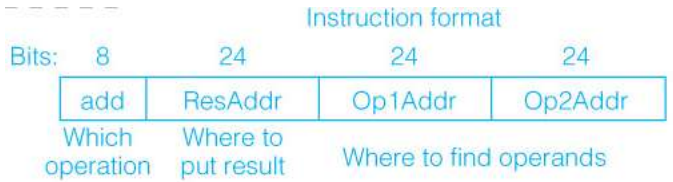


Fig 2.4 The 3 Address Instruction

- Address of next instruction kept in a processor state register—the PC (Except for explicit Branches/Jumps)
- Rest of addresses in instruction



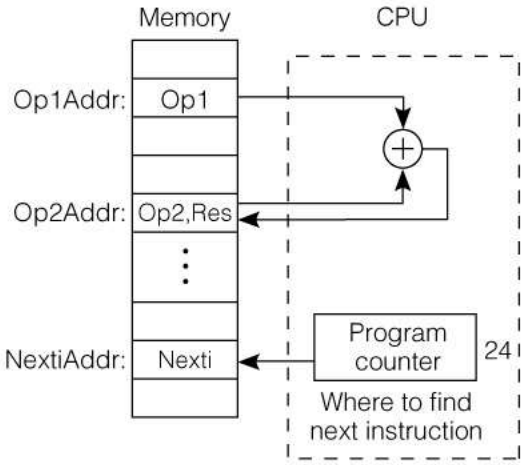
3 Address Example



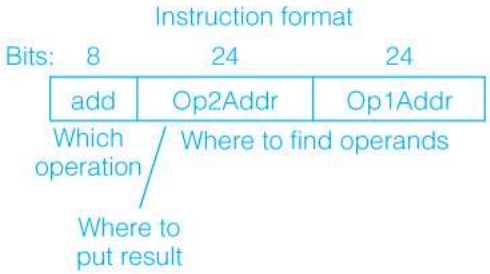
- Instructions in memory
  - Each address = 3 bytes (1 word)
  - Total bytes for ALU instruction = 3 x 3 + 1 = 10
  - Total words = 3 x 1 + 1 = 4
- Memory access
  - Instruction fetch = 4 words
  - Operands = 2 words (read)
  - Result = 1 word (write)
  - Total = 4 + 2 + 1 = 7

Fig. 2.5 The 2 Address Instruction

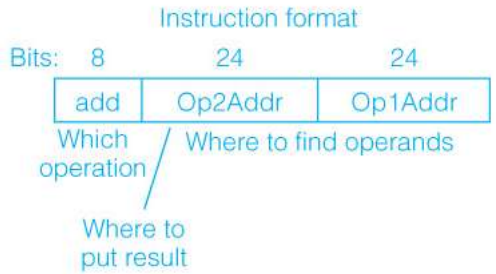
- Result overwrites operand 2
- Needs only 2 addresses in the instruction but less choice in placing data



add Op2, Op1 (Op2 ← Op2 + Op1)



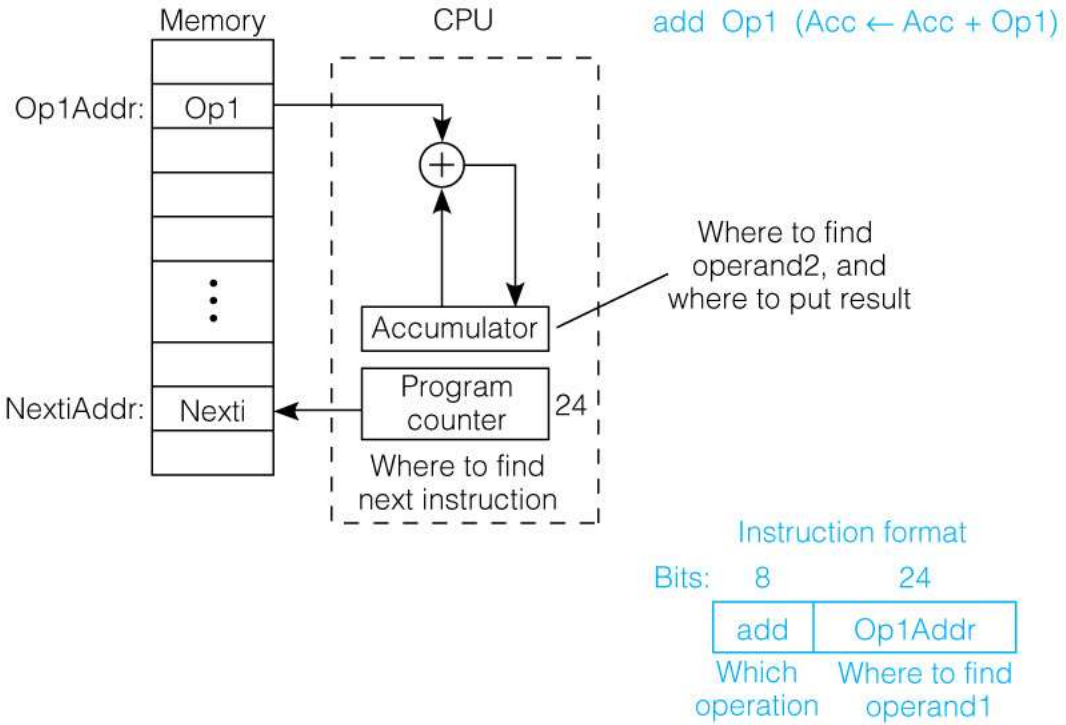
2 Address Example



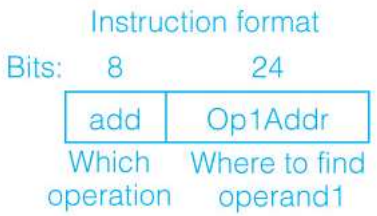
- Instructions in memory
  - Each address = 3 bytes (1 word)
  - Total bytes for ALU instruction = 2 x 3 + 1 = 7
  - Total words = 2 x 1 + 1 = 3
- Memory access
  - Instruction fetch = 3 words
  - Operands = 2 words (read)
  - Result = 1 word (write)
  - Total = 3 + 2 + 1 = 6

Fig. 2.6 1 Address Instructions

- Special CPU, the accumulator, supplies 1 operand and stores result
- One memory address used for other operand



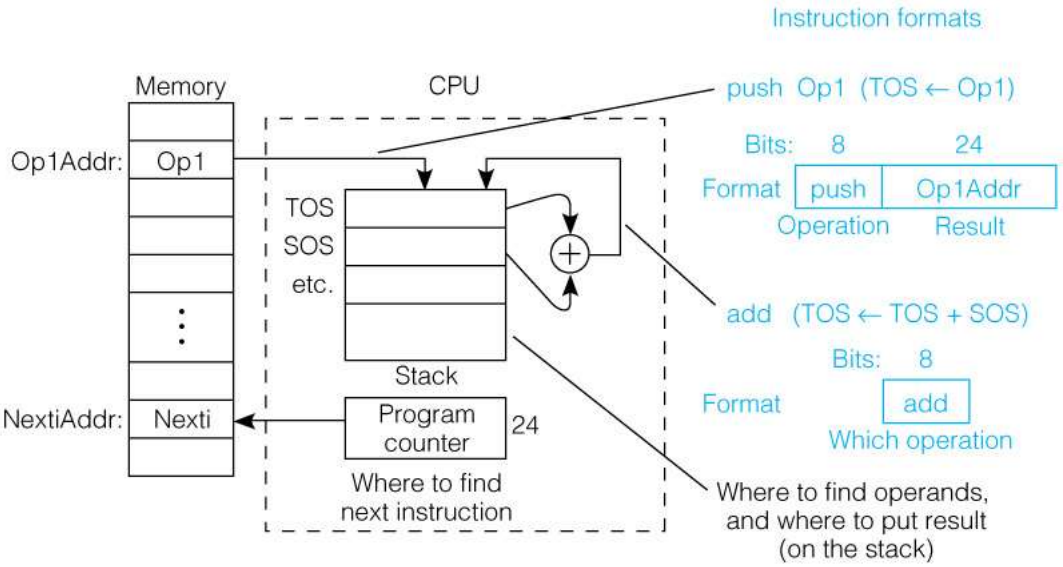
Accumulator Example



- Instructions in memory
  - Each address = 3 bytes (1 word)
  - Total bytes for ALU instruction = 1 x 3 + 1 = 4
  - Total words = 1 x 1 + 1 = 2
- Memory access
  - Instruction fetch = 2 words
  - Operands = 1 words (read)
  - Total = 2 + 1 = 3 (wrong!)
  - Must load/store contents of accumulator
    - `lda OpAddr` `sta OpAddr`
  - Total = 3 + accumulator access (e.g. 4 or 5)

Fig. 2.7 The 0 Address Instruction

- Uses a push down stack in CPU
- Arithmetic uses stack for both operands. The result replaces them on the TOS
- Computer must have a 1 address instruction to push and pop operands to and from the stack



Stack Example

- Instructions in memory
  - Each address = 3 bytes (1 word)
  - Total bytes for ALU instruction = 1 = 1
  - Total words = 1
    - Single add op is usually not sufficient, require multiple instructions to complete
- Memory access
  - Instruction fetch = 1 words
  - Need to manage contents on top of stack
    - `push pop`
  - Total = 1 + push/pop (e.g. 2, 3, 4)
- Extra instructions are required to get data to the stack

Example 2.1

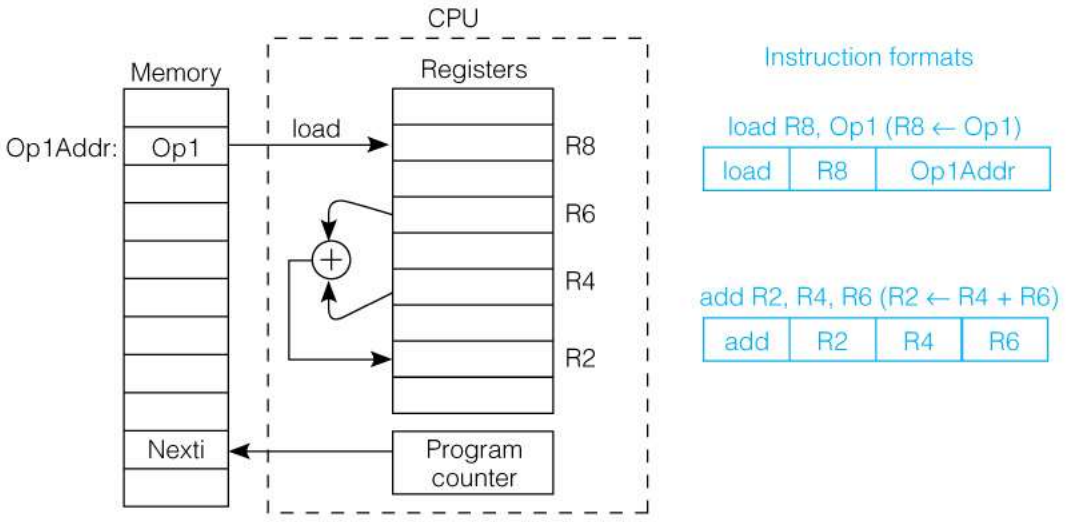
- Evaluate `a = (b+c)*d-e`
- for 3- 2- 1- and 0-address machines

3-Address	2-Address	1-Address	0-Address

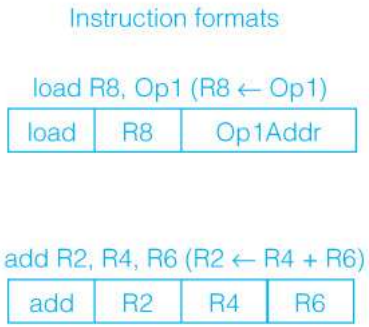
- How many instructions and memory addresses are needed?

Fig. 2.8 General Register Machines

- Most common choice for general purpose computers
- Registers specified by “small” address (3 to 6 bits for 8 to 64 registers)
  - Close to CPU for speed and reuse for complex operations



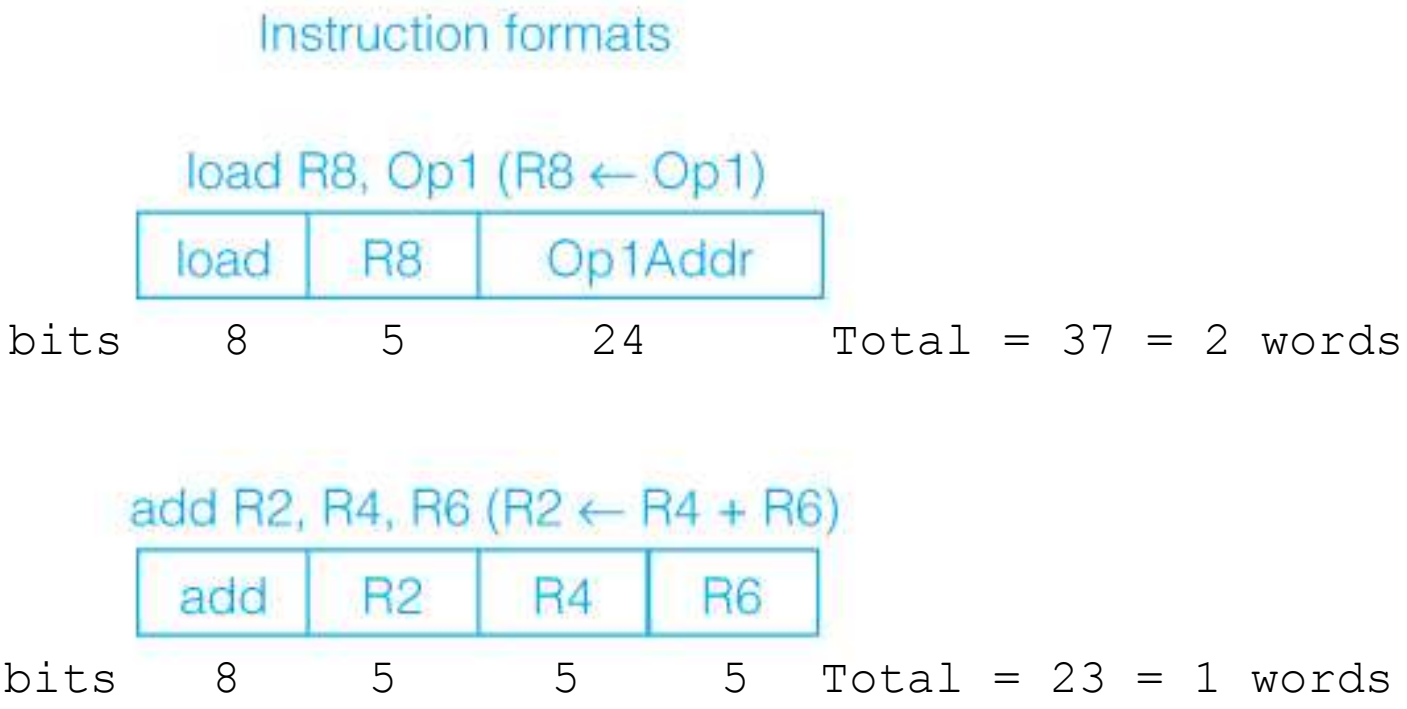
1-1/2 Address Instructions



- "Small" register address = half address
- 1-1/2 addresses
  - Load/store have one long & one short address
  - 2-operand arithmetic instruction has 3 half addresses

General Register Example

- More complex encoding
  - Assume 32 registers = 5 bits to specify



Real Machines

- General registers offer greatest flexibility
  - Possible because of low price of memory
- Most real machines have a mixture of 3, 2, 1, 0, 1-1/2 address instructions
  - A distinction can be made on whether arithmetic instructions use data from memory
- Load-store machine
  - Registers used for operands and results of ALU instructions
  - Only load and store instructions reference memory
- Other machines have a mix of register-memory and memory-memory instructions

Instructions/Register Trade-Offs

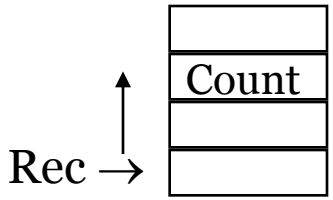
- 3-address machines have shortest code but large number of bits per instruction
- 0-address machines have longest code but small number of bits per instruction
  - Still require 1-address (push, pop) instructions
- General register machines use short internal register addresses in place of long memory addresses
- Load-store machines only allow memory addresses in data movement instructions (load, store)
- Register access is much faster than memory access
- Short instructions are faster

Addressing Modes

- Addressing mode is hardware support for a useful way of determining a memory address
- Different addressing modes solve different HLL problems
  - Some addresses may be known at compile time, e.g. global vars.
  - Others may not be known until run time, e.g. pointers
  - Addresses may have to be computed
    - Record (struct) components:
      - variable base(full address) + const.(small)
    - Array components:
      - const. base(full address) + index var.(small)
- Possible to store constant values without using another memory cell by storing them with or adjacent to the instruction itself.

HLL Examples of Structured Addresses

- C language: rec -> count
  - rec is a pointer to a record: full address variable
  - count is a field name: fixed byte offset, say 24
- C language: v[i]
  - v is fixed base address of array: full address constant
  - i is name of variable index: no larger than array size
- Variables must be contained in registers or memory cells
- Small constants can be contained in the instruction
- Result: need for "address arithmetic."
  - E.g. Address of Rec -> Count is address of Rec + offset of count.



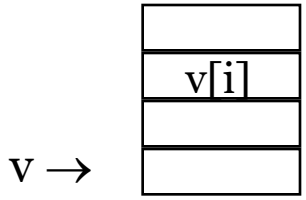


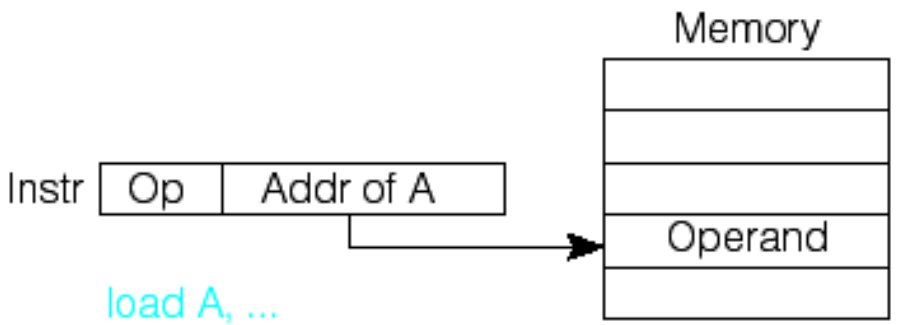
Fig 2.9 Common Addressing Modes a-d

(a) Immediate addressing:

instruction contains the operand

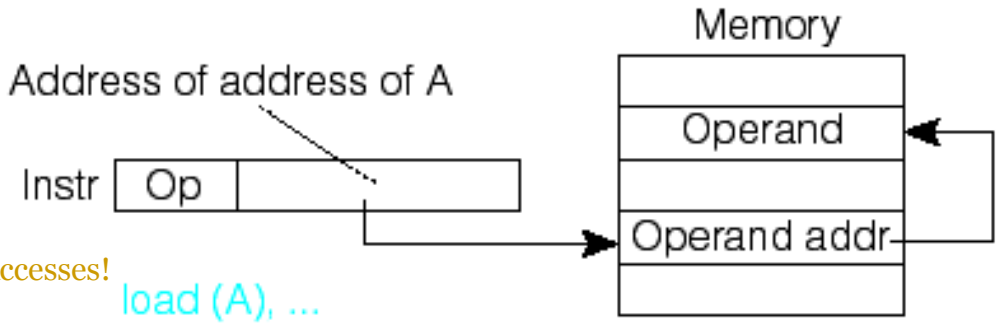
(b) Direct addressing:

instruction contains address of operand



(c) Indirect addressing:

instruction contains address of address of operand



Two Memory Accesses!

(d) Register direct addressing:

register contains operand

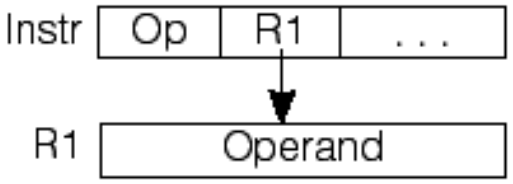
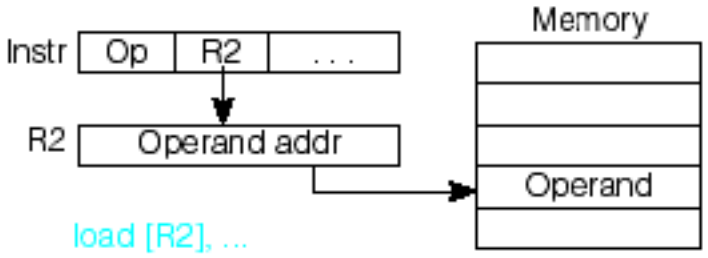


Fig 2.9 Common Addressing Modes e-g

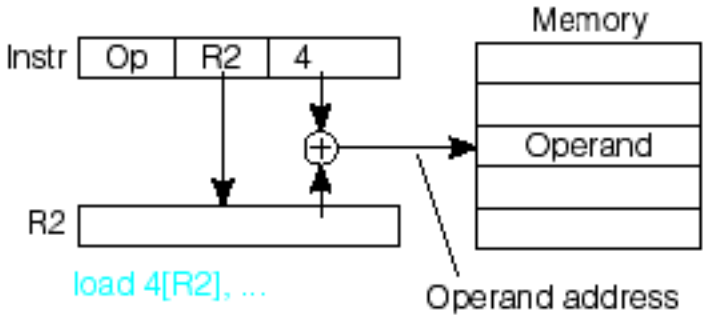
(e) Register indirect addressing:

register contains address of operand



(f) Displacement (based or indexed) addressing:

address of operand = register + constant



(g) Relative addressing:

address of operand = PC + constant