Chapter 2: Machines, Machine Languages, and Digital Logic

Topics

What Must an Instruction Specify?

Data Flow ←

- Which operation to perform — add r0, r1, r3
  - Ans: Op code: add, load, branch, etc.
- Where to find the operand or operands — add r0, r1, r3
  - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result — add r0, r1, r3
  - Again CPU register or memory cell
- Location of next instruction — add r0, r1, r3
br endloop
  - Almost always memory cell pointed to by program counter—PC
- Sometimes there *is* no operand, or no result, or no next instruction. Can you think of examples?

Instructions Can Be Divided into 3 Classes

- 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
  - Change the form of one or more operands to produce a result stored in another location
  - Add, Sub, Shift, etc.
- Branch instructions (control flow instructions)
  - Alter the normal flow of control from executing the next instruction in sequence
  - Br Loc, Brz Loc2,—unconditional or conditional branches

Tbl 2.1 Examples of Data Movement Instructions

Instruction	Meaning	Machine
`MOV A, B`	Move 16 bits from memory location `A` to Location `B`	VAX11
`LDA A, Addr`	Load accumulator `A` with the byte at memory location `Addr`	M6800
`lwz R3, A`	Move 32-bit data from memory location A to register `R3`	PPC601
`li $3, 455`	Load the 32-bit integer 455 into register `$3`	MIPS R3000
`mov R4, dout`	Move 16-bit data from `R4` to output port `dout`	DEC PDP11
`IN, AL, KBD`	Load a byte from in port `KBD` to accumulator	Intel Pentium
`LEA.L (A0), A2`	Load the address pointed to by `A0` into `A2`	M6800

- Lots of variation, even with one instruction type

Tbl 2.2 Examples of ALU Instructions

Instruction	Meaning	Machine
MULF A, B, C	**multiply the 32-bit floating point values at mem loc'ns. A and B, store at C**	**VAX11**
nabs r3, r1	**Store abs value of r1 in r3**	**PPC601**
ori $2, $1, 255	**Store logical OR 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 bit pos'ns.**	**Intel 8086**

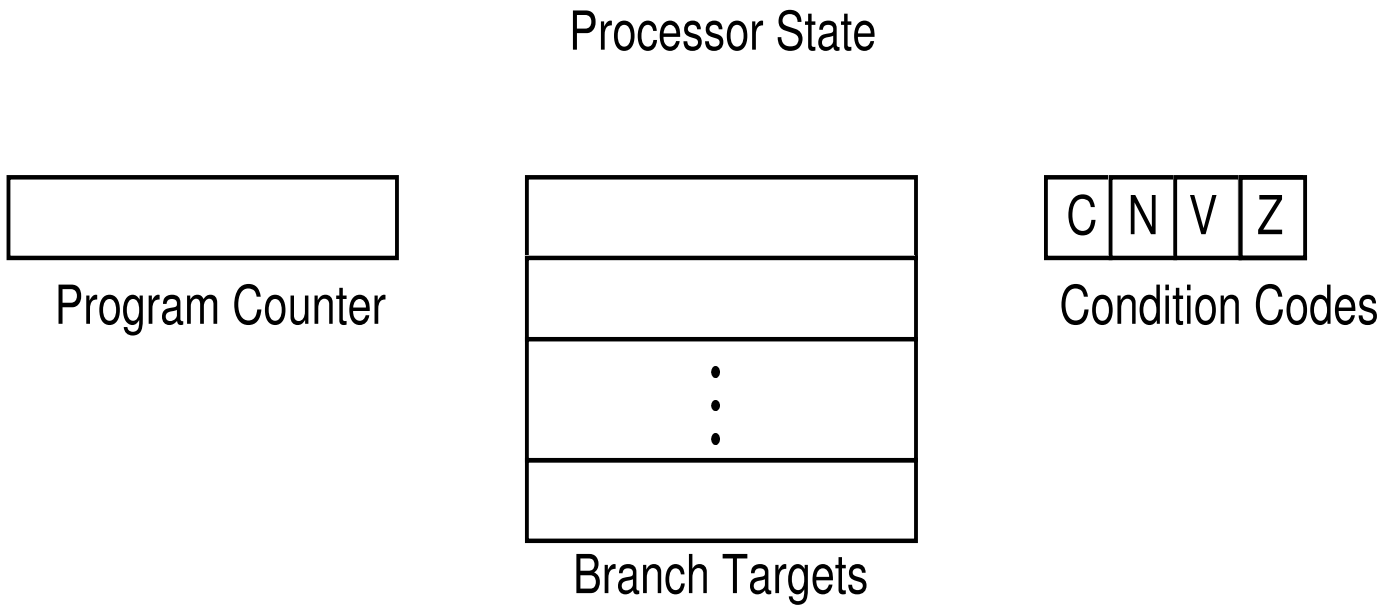
- **Notice again the complete dissimilarity of both syntax and semantics.**

Tbl 2.3 Examples of Branch Instructions

Instruction	Meaning	Machine
BLSS A, Tgt	**Branch to address Tgt if the least significant bit of mem loc'n. A is set (i.e. = 1)**	**VAX11**
bun r2	**Branch to location in R2 if result of previous floating point computation was Not a Number (NAN)**	**PPC601**
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 $\neq$ 0**	**DEC PDP11**
JCXZ Addr	**Jump to Addr if contents of register CX $\neq$ 0.**	**Intel 8086**

CPU Registers Associated with Flow of Control—Branch Instructions

- Program counter usually locates next instruction
- Condition codes may control branch
- Branch targets may be separate registers



HLL Conditionals Implemented by Control Flow Change

- **Conditions are computed by arithmetic instructions**
- **Program counter is changed to execute only instructions associated with true conditions**

C language

```
if NUM==5 then SET=7
```

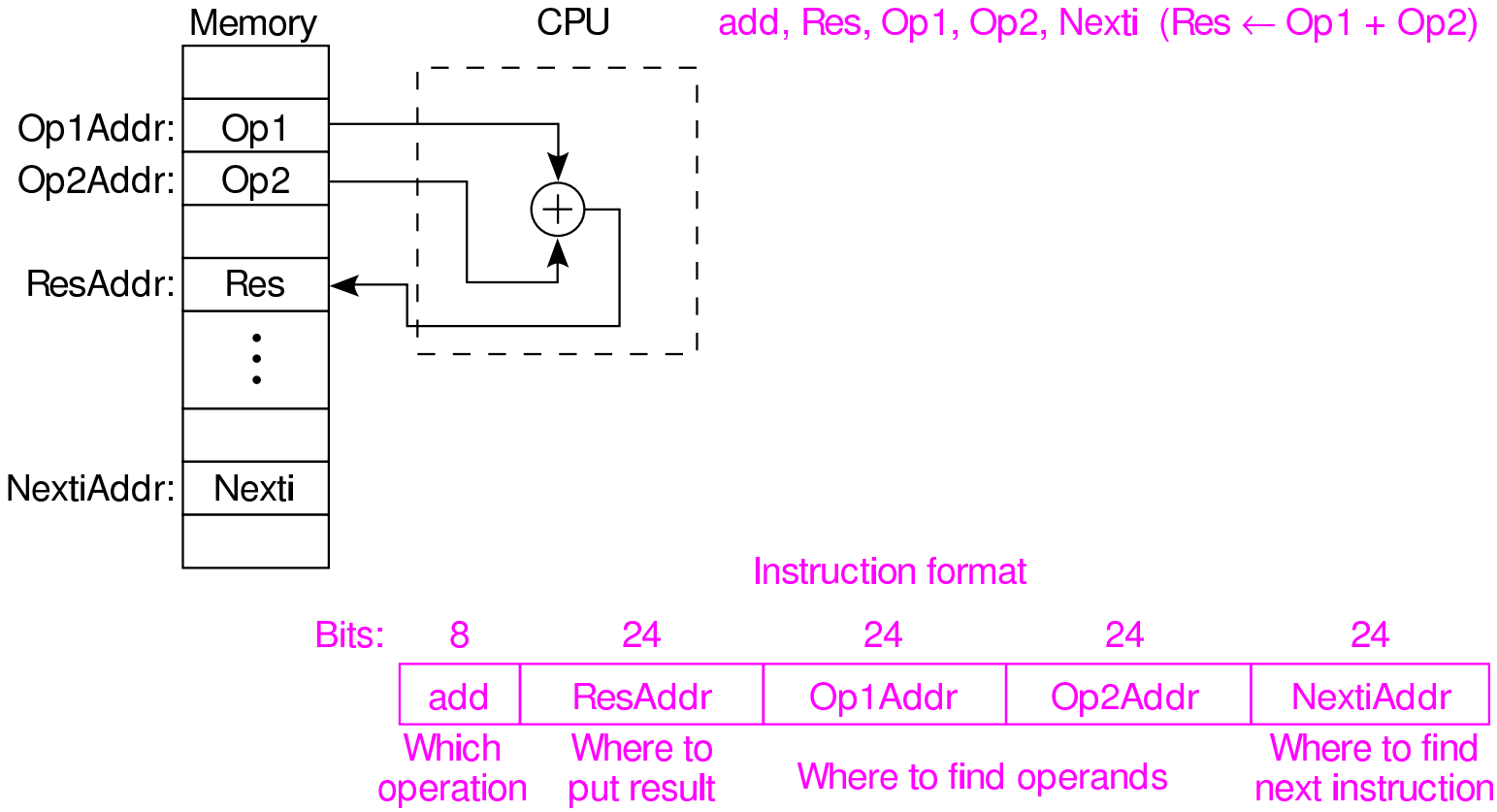
Assembly language

```
      CMP.W   #5, NUM   ;the comparison
      BNE     L1        ;conditional branch
      MOV.W   #7, SET   ;action if true
L1    ...               ;action if false
```

3-, 2-, 1-, & 0-Address ISAs

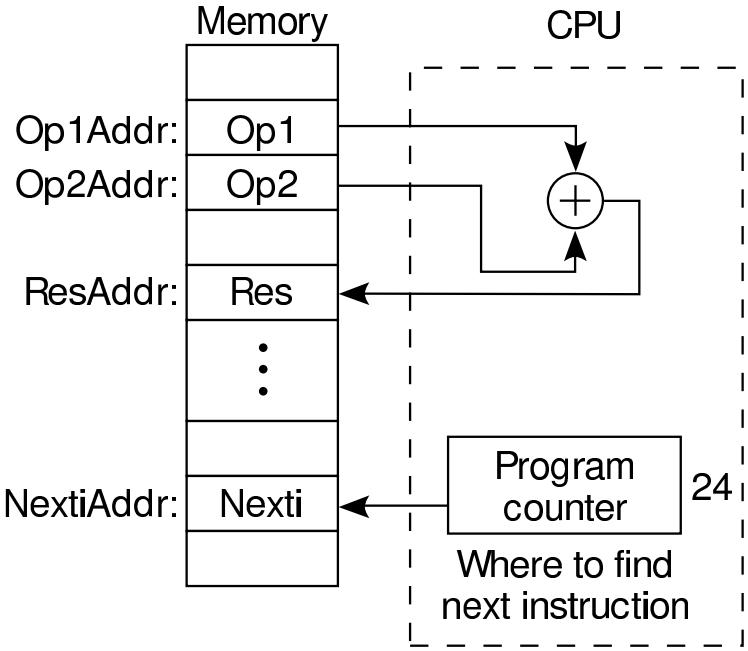
- The classification is based on arithmetic instructions that have two operands and one result
- The key issue is "how many of these are specified by memory addresses, as opposed to being specified implicitly"
- A 3-address instruction specifies memory addresses for both operands and the result R ← Op1 op Op2
- A 2-address instruction overwrites one operand in memory with the result Op2 ← Op1 op Op2
- A 1-address instruction has a processor, called the accumulator register, to hold one operand & the result (no addr. needed) Acc ← Acc op Op1
- A 0-address + uses a CPU register stack to hold both operands and the result TOS ← TOS op SOS (where TOS is Top Of Stack, SOS is Second On Stack)
- The 4-address instruction, hardly ever seen, also allows the address of the next instruction to specified explicitly

Fig 2.2 The 4-Address Machine and Instruction Format



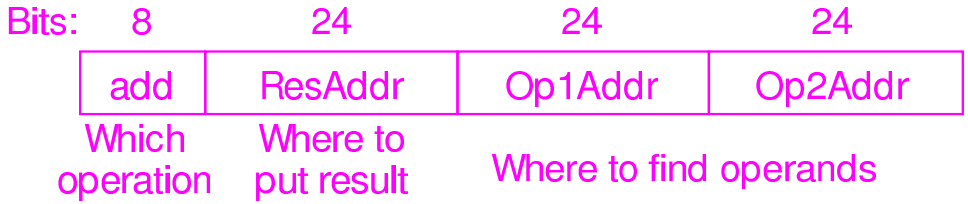
- **Explicit addresses for operands, result, & next instruction**
- **Example assumes 24-bit addresses**
  - **Discuss: size of instruction in bytes**

Fig 2.3 The 3-Address Machine and Instruction Format



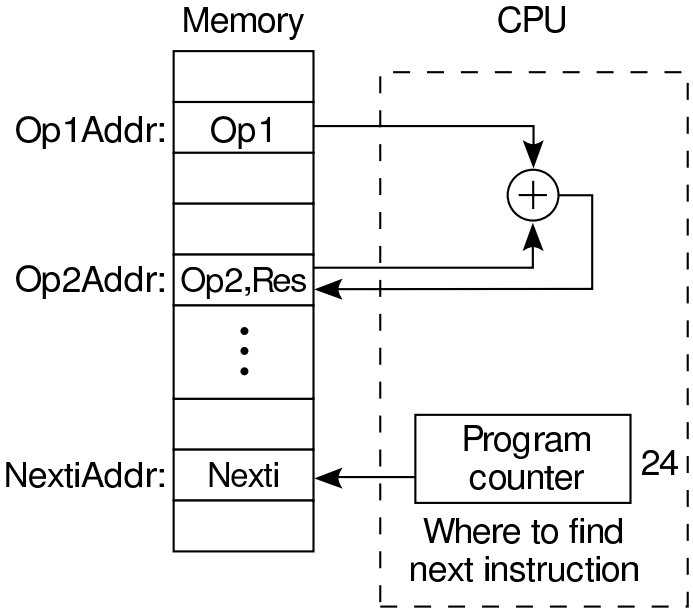
add, Res, Op1, Op2 (Res ← Op2 + Op1)

Instruction format

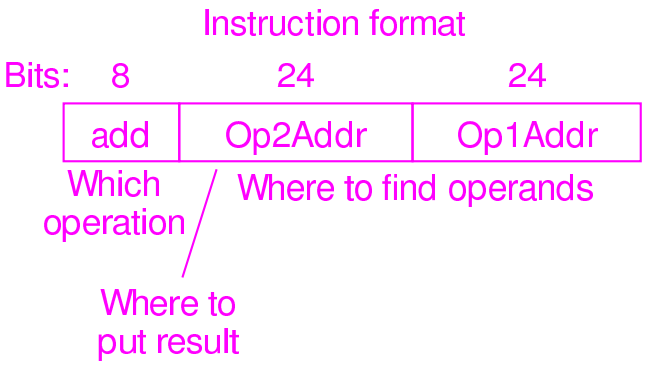


- **Address of next instruction kept in processor state register—the PC (except for explicit branches/jumps)**
- **Rest of addresses in instruction**
  - **Discuss: savings in instruction word size**

Fig 2.4 The 2-Address Machine and Instruction Format

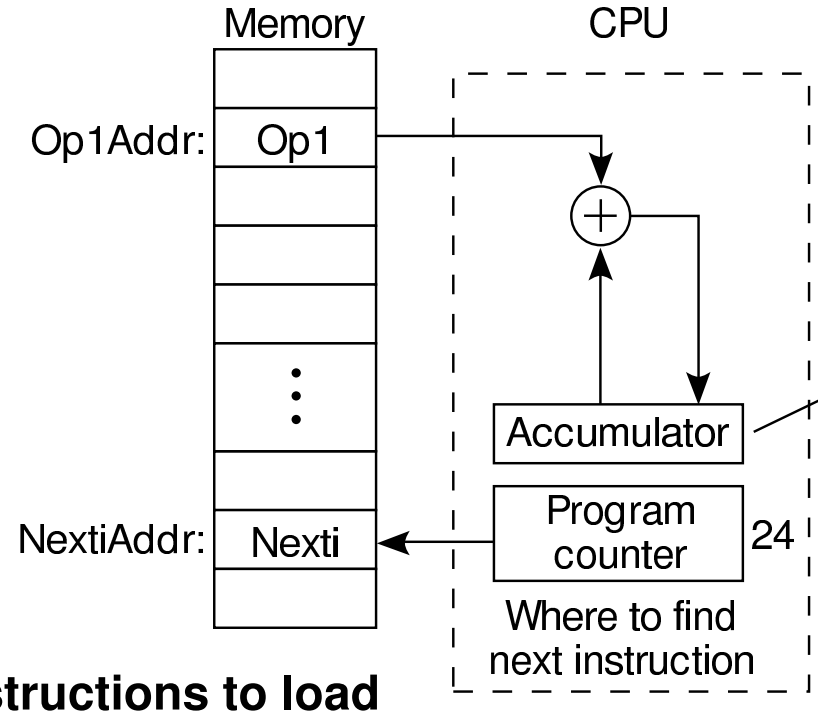


add Op2, Op1 (Op2 ← Op2 + Op1)



- **Result overwrites Operand 2**
- **Needs only 2 addresses in instruction but less choice in placing data**

Fig 2.5 1-Address Machine and Instruction Format



add Op1 (Acc ← Acc + Op1)

Where to find operand2, and where to put result

Need instructions to load and store operands:
LDA OpAddr
STA OpAddr

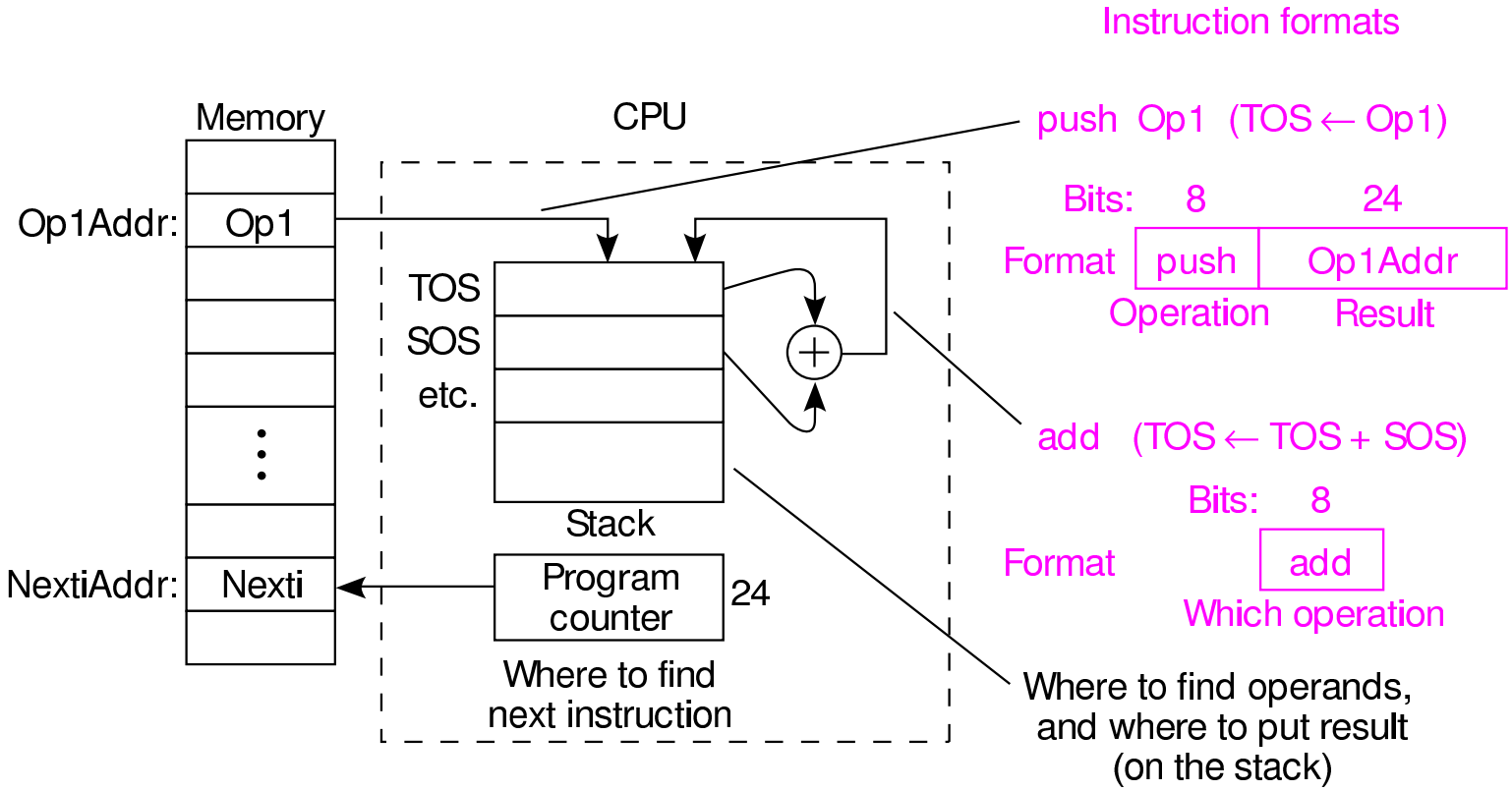
Instruction format

Bits: 8 24

add	Op1Addr
Which operation	Where to find operand1

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

Fig 2.6 The 0-Address, or Stack, Machine and Instruction Format



- **Uses a push-down stack in CPU**
- **Arithmetic uses stack for both operands and the result**
- **Computer must have a 1-address instruction to push and pop operands to and from the stack**

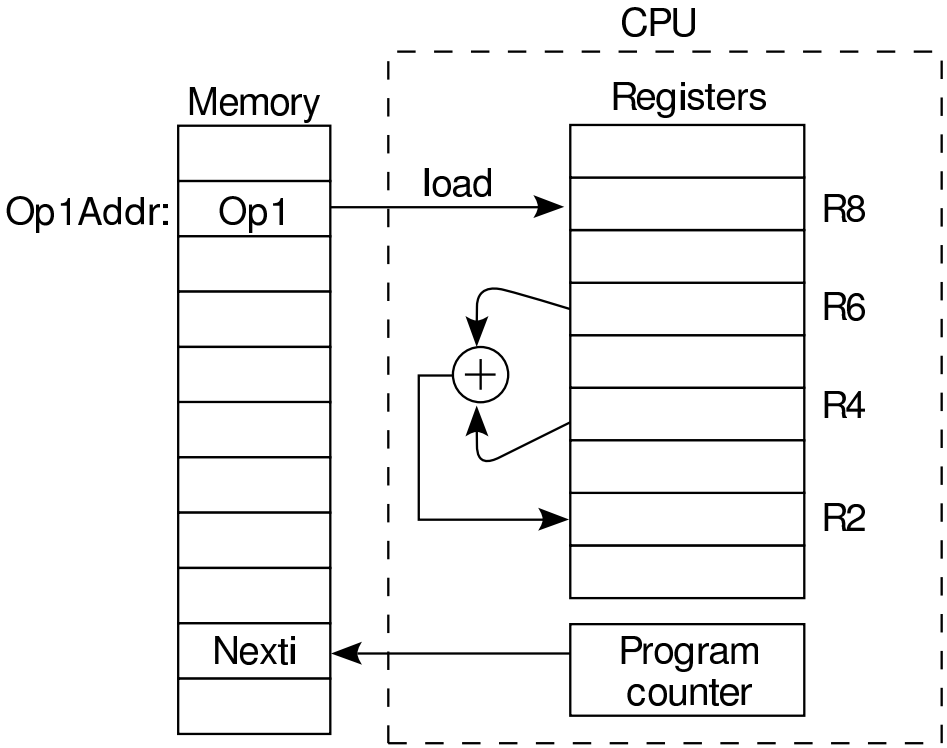
Example 2.1 Expression Evaluation for 3-, 2-, 1-, and 0-Address Machines

Evaluate a = (b+c)*d - e

3-address	2-address	1-address	Stack
add a, b, c	load a, b	load b	push b
mpy a, a, d	add a, c	add c	push c
sub a, a, e	mpy a, d	mpy d	add
	sub a, e	sub e	push d
		store a	mpy
			push e
			sub
			pop a

- **Number of instructions & number of addresses both vary**
- **Discuss as examples: size of code in each case**

Fig 2.7 General Register Machine and Instruction Formats



Instruction formats

load R8, Op1 (R8 ← Op1)

load	R8	Op1Addr

add R2, R4, R6 (R2 ← R4 + R6)

add	R2	R4	R6

- **It is the most common choice in today's general-purpose computers**
- ***Which* register is specified by small "address" (3 to 6 bits for 8 to 64 registers)**
- **Load and store have one long & one short address: 1-1/2 addresses**
- **Arithmetic instruction has 3 "half" addresses**