Computer Architecture and Operating System

Prof. Indranil Sengupta

Department of Computer Science and Engineering

IIT Kharagpur

Instruction Set Architecture

Introduction

- Instruction Set Architecture (ISA)
  - Serves as an interface between software and hardware.
  - Typically consists of information regarding the programmer's view of the architecture (i.e. the registers, address and data buses, etc.).
  - Also consists of the instruction set.
- Many ISA's are not specific to a particular computer architecture.
  - They survive across generations.
  - Classic examples: IBM 360 series, Intel x86 series, etc.

Instruction Set Design Issues

- Number of explicit operands:
  - 0, 1, 2 or 3.
- Location of the operands:
  - Registers, accumulator, memory.
- Specification of operand locations:
  - Addressing modes: register, immediate, indirect, relative, etc.
- Sizes of operands supported:
  - Byte (8-bits), Half-word (16-bits), Word (32-bits), Double (64-bits), etc.
- Supported operations:
  - ADD, SUB, MUL, AND, OR, CMP, MOVE, JMP, etc.

Evolution of Instruction Sets

1.	Accumulator based:	1960's	(EDSAC, IBM 1130)
2.	Stack based:	1960-70	(Burroughs 5000)
3.	Memory-Memory based:	1970-80	(IBM 360)
4.	Register-Memory based:	1970-present	(Intel x86)
5.	Register-Register based:	1960-present	(MIPS, CDC 6600, SPARC)

1: 1-address instructions:
ADD X → ACC = ACC + Mem[X]

2: 0-address instructions:
ADD → TOS = TOS + NEXT

3: 2- or 3-address instructions:
ADD A,B → Mem[A] = Mem[A] + Mem[B]
ADD A,B,C → Mem[A] = Mem[B] + Mem[C]

4: 2-address instructions:
LOAD R1,X → R1 = Mem[X]

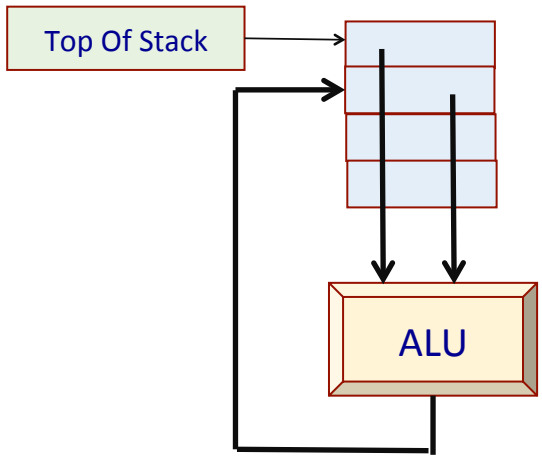
5: 3-address instructions:
ADD R1,R2,R3 → R1 = R2 + R3

Example Code Sequence for Z = X + Y

a) **Stack machine:**

PUSH X
PUSH Y
ADD
POP Z

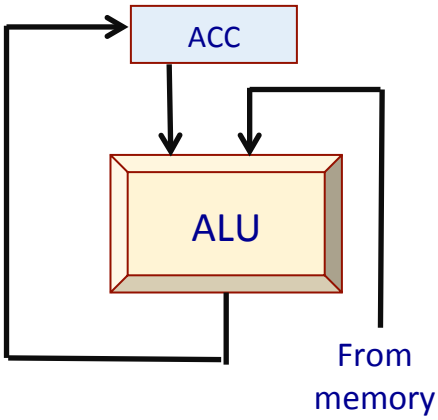
- The ADD instruction pops two elements from stack, adds them, and pushes back result.



b) Accumulator based machine:

```
LOAD   X     // ACC = Mem[X]
ADD    Y     // ACC = ACC + Mem[Y]
STORE  Z     // Mem[Z] = ACC
```

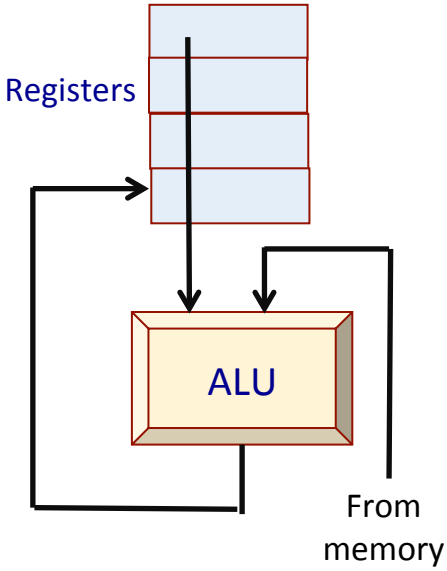
- All instructions assume that one of the operands (and also the result) is in a special register called *accumulator*.



c) Register-Memory machine:

```
LOAD   R2,X    // R2 = Mem[X]
ADD    R2,Y    // R2 = R2 + Mem[Y]
STORE  Z,R2    // Mem[Z] = R2
```

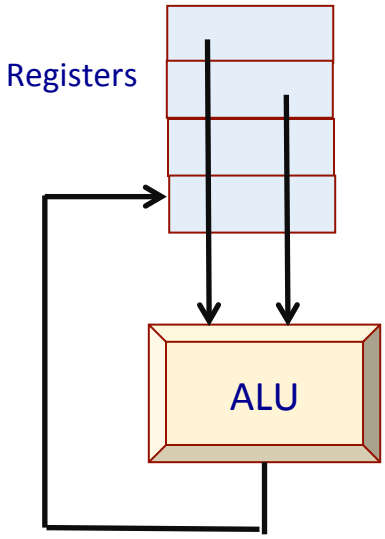
- One of the operands is assumed to be in register and another in memory.



d) **Register-Register machine:**

```
LOAD   R1,X        // R1 = Mem[X]
LOAD   R2,Y        // R2 = Mem[Y]
ADD    R3,R1,R2    // R3 = R1 + R2
STORE  Z,R3        // Mem[Z] = R3
```

- Also called *load-store architecture*, as only LOAD and STORE instructions can access memory.



About General Purpose Registers (GPRs)

- Older architectures had a large number of special purpose registers.
  - Program counter, stack pointer, index register, flag register, accumulator, etc.
- Newer architectures, in contrast, have a large number of GPRs.
- Why?
  - Easy for the compiler to assign some variables to registers.
  - Registers are much faster than memory.
  - More compact instruction encoding as fewer bits are required to specify registers.
  - Many processors have 32 or more GPR's.

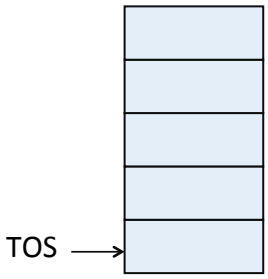
COMPARISON BETWEEN VARIOUS ARCHITECTURE TYPES

(a) Stack Architecture

- Typical instructions:

  PUSH X, POP X

  ADD, SUB, MUL, DIV



- **Example: Y = A / B – (A – C * B)**

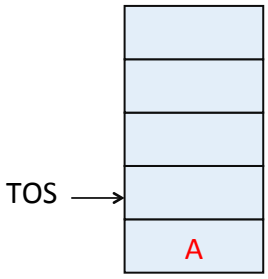
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

(a) Stack Architecture

- Typical instructions:

  PUSH X, POP X

  ADD, SUB, MUL, DIV



- **Example: Y = A / B – (A – C * B)**

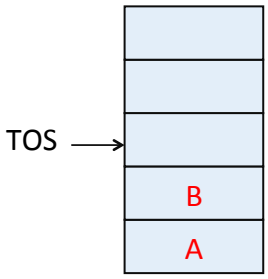
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

(a) Stack Architecture

- Typical instructions:

  PUSH X, POP X

  ADD, SUB, MUL, DIV



- **Example: Y = A / B – (A – C * B)**

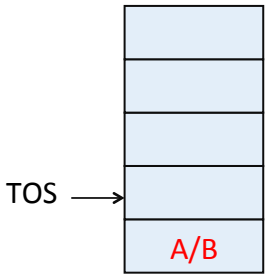
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

(a) Stack Architecture

- Typical instructions:

  PUSH X, POP X

  ADD, SUB, MUL, DIV



- **Example: Y = A / B – (A – C * B)**

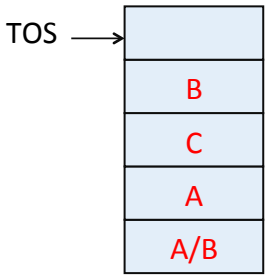
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

(a) Stack Architecture

- Typical instructions:

  PUSH X, POP X

  ADD, SUB, MUL, DIV



- **Example: Y = A / B – (A – C * B)**

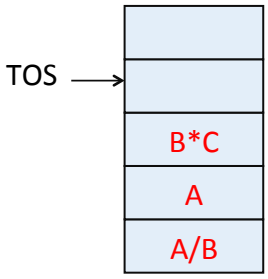
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

(a) Stack Architecture

- Typical instructions:

  PUSH X, POP X

  ADD, SUB, MUL, DIV



- **Example: Y = A / B – (A – C * B)**

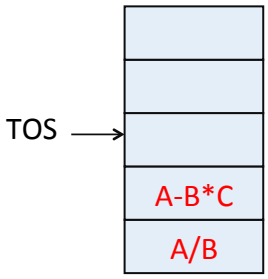
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

(a) Stack Architecture

- Typical instructions:

  PUSH X, POP X

  ADD, SUB, MUL, DIV



- **Example: Y = A / B – (A – C * B)**

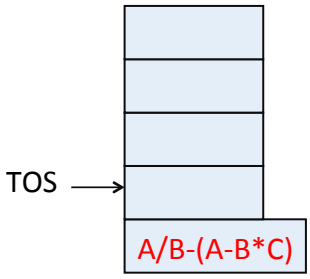
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

(a) Stack Architecture

- Typical instructions:

  PUSH X, POP X

  ADD, SUB, MUL, DIV



- **Example: Y = A / B – (A – C * B)**

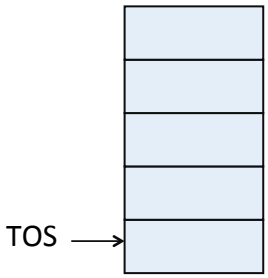
  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y

(a) Stack Architecture

- Typical instructions:

  PUSH X, POP X

  ADD, SUB, MUL, DIV



- **Example: Y = A / B – (A – C * B)**

  PUSH A
  PUSH B
  DIV
  PUSH A
  PUSH C
  PUSH B
  MUL
  SUB
  SUB
  POP Y Y = RESULT

(b) Accumulator Architecture

- Typical instructions:

  LOAD X, STORE X

  ADD X, SUB X, MUL X, DIV X

Example: Y = A / B – (A – C * B)

```
LOAD   C
MUL    B
STORE  D      // D = C * B
LOAD   A
SUB    D
STORE  D      // D = A – C * B
LOAD   A
DIV    B
SUB    D
STORE  Y
```


(c) Memory-Memory Architecture

- Typical instructions (3 operands):

  ADD X,Y,Z

  SUB X,Y,Z

  MUL X,Y,Z

- Typical instructions (2 operands):

  MOV X,Y

  ADD X,Y

  SUB X,Y

  MUL X,Y

Example: Y = A / B – (A – C * B)

```
DIV    D,A,B
MUL    E,C,B
SUB    E,A,E
SUB    Y,D,E
```

```
MOV    D,A
DIV    D,B
MOV    E,C
MUL    E,B
SUB    A,E
SUB    D,A
```

(d) Load-Store Architecture

- Typical instructions:

```
LOAD   R1,X
STORE  Y,R2
ADD    R1,R2,R3
SUB    R1,R2,R3
```

Example: Y = A / B – (A – C * B)

```
LOAD   R1,A
LOAD   R2,B
LOAD   R3,C
DIV    R4,R1,R2
MUL    R5,R3,R2
SUB    R5,R1,R5
SUB    R4,R4,R5
STORE  Y,R4
```


Instruction Format

Instruction Format

- An instruction consists of two parts:-
  - *Operation Code or Opcode*
    - Specifies the operation to be performed by the instruction.
    - Various categories of instructions: data transfer, arithmetic and logical, control, I/O and special machine control.
  - *Operand(s)*
    - Specifies the source(s) and destination of the operation.
    - Source operand can be specified by an immediate data, by naming a register, or specifying the address of memory.
    - Destination can be specified by a register or memory address.

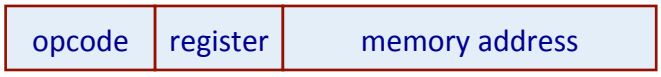
- Number of operands varies from instruction to instruction.
- Also for specifying an operand, various *addressing modes* are possible:
  - Immediate addressing
  - Direct addressing
  - Indirect addressing
  - Relative addressing
  - Indexed addressing, and many more.

Instruction Format Examples

Implied addressing: NOP, HALT, ADD

1-address: ADD X, LOAD M

2-address: ADD X,Y



Register-memory: ADD R1,X

Register-register: ADD R1,R2,R3

ADDRESSING MODES

What are Addressing Modes?

- They specify the mechanism by which the operand data can be located.
- Some ISA's are quite complex and supports many addressing modes.
- ISA's based on load-store architecture are usually simple and support very limited number of addressing modes.
- Various addressing modes exist:
  - Immediate, Direct, Indirect, Register, Register Indirect, Indexed, Stack, Relative, Base, etc.
  - Not all processors support all addressing modes.

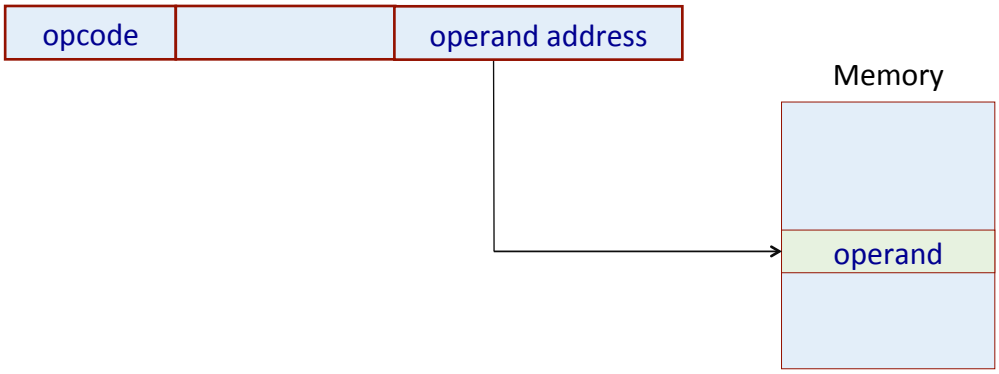
(a) Immediate Addressing

- The operand is part of the instruction itself.
  - No memory reference is required to access the operand.
  - Fast but limited range (because a limited number of bits are provided to specify the immediate data).
- Examples:
  - ADD #25 // ACC = ACC + 25
  - ADD R1,R2,#42 // R1 = R2 + 42

(b) Direct Addressing

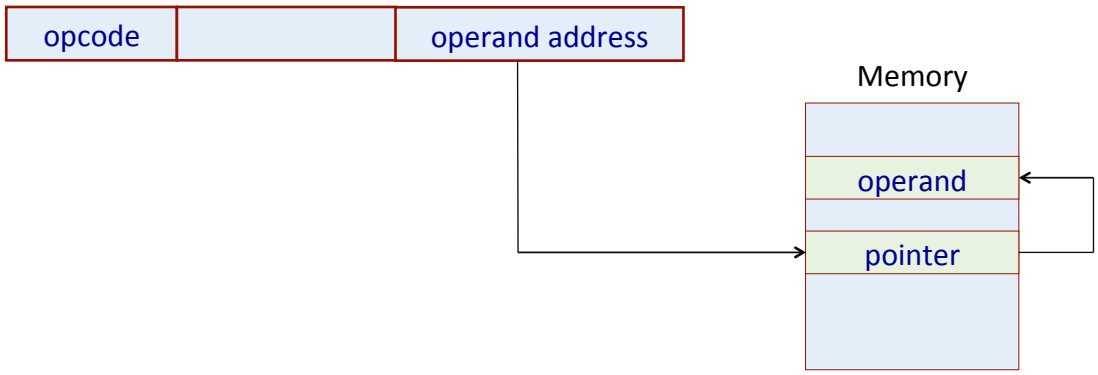
- The instruction contains a field that holds the memory address of the operand.

- Example:
  - ADD R1,20A6H // R1 = R1 + Mem[20A6]
- Single memory access is required to access the operand.
  - No additional calculations required to determine the operand address.
  - Limited address space (as number of bits is limited, say, 16 bits).



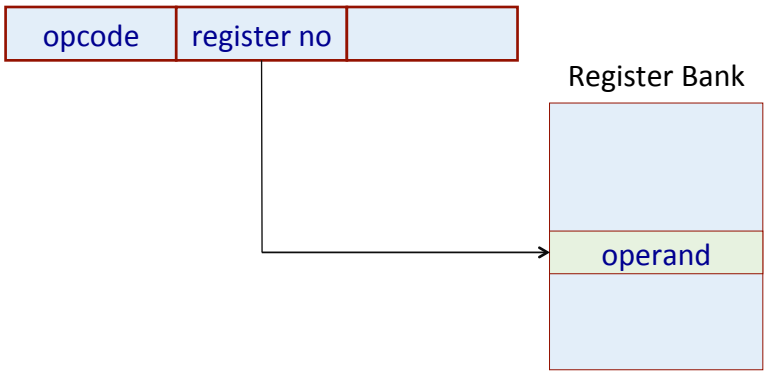
(c) Indirect Addressing

- The instruction contains a field that holds the memory address, which in turn holds the memory address of the operand.
- Two memory accesses are required to get the operand value.
- Slower but can access large address space.
  - Not limited by the number of bits in operand address like direct addressing.
- Examples:
  - ADD R1,(20A6H) // R1 = R1 + (Mem[20A6])



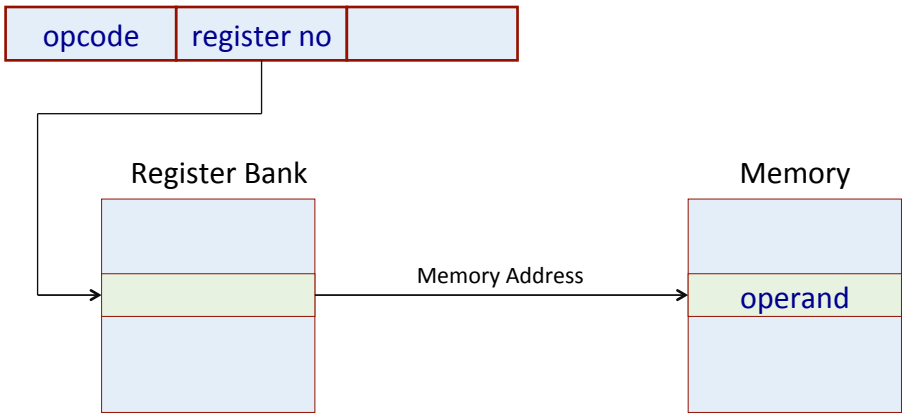
(d) Register Addressing

- The operand is held in a register, and the instruction specifies the register number.
  - Very few number of bits needed, as the number of registers is limited.
  - Faster execution, since no memory access is required for getting the operand.
- Modern load-store architectures support large number of registers.
- Examples:
  - ADD R1,R2,R3 // R1 = R2 + R3
  - MOV R2,R5 // R2 = R5



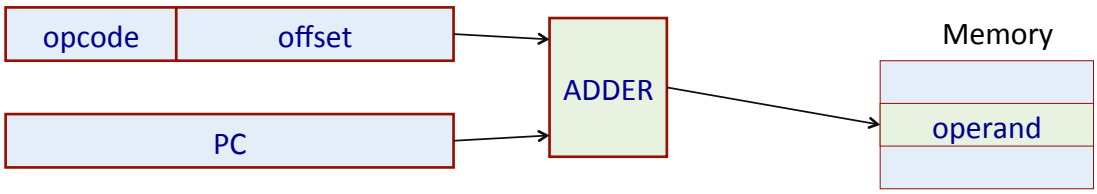
(e) Register Indirect Addressing

- The instruction specifies a register, and the register holds the memory address where the operand is stored.
  - Can access large address space.
  - One fewer memory access as compared to *indirect addressing*.
- Example:
  - ADD R1,(R5) // R1 = R1 + Mem[R5]



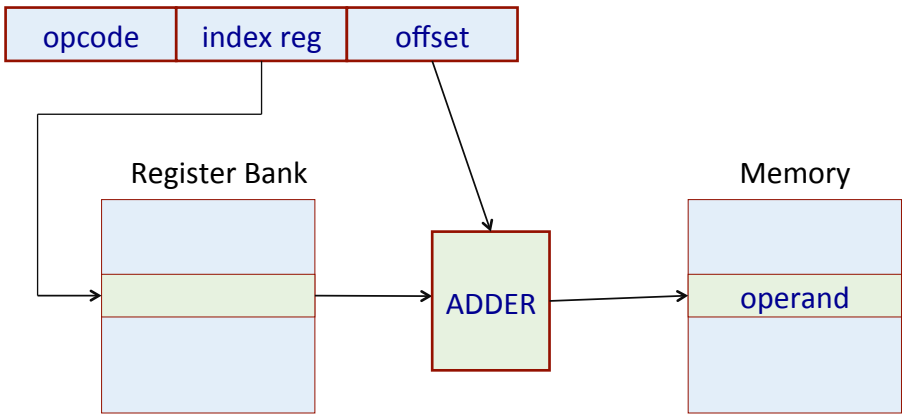
(f) Relative Addressing (PC Relative)

- The instruction specifies an *offset* or *displacement*, which is added to the program counter (PC) to get the effective address of the operand.
  - Since the number of bits to specify the offset is limited, the range of relative addressing is also limited.
  - If a 12-bit offset is specified, it can have values ranging from -2048 to +2047.



(g) Indexed Addressing

- Either a special-purpose register, or a general-purpose register, is used as *index register* in this addressing mode.
- The instruction specifies an *offset* or *displacement*, which is added to the index register to get the effective address of the operand.
- Example:
  - LOAD R1,1050(R3) // R1 = Mem[1050+R3]
- Can be used to sequentially access the elements of an array.
  - Offset gives the starting address of the array, and the index register value specifies the array element to be used.



(h) Stack Addressing

- Operand is implicitly on top of the stack.
- Used in zero-address machines earlier.
- Examples:
  - ADD
  - PUSH X
  - POP X
- Many processors have a special register called the *stack pointer (SP)* that keeps track of the stack-top in memory.
  - PUSH, POP, CALL, RET instructions automatically modify SP.

Some Other Addressing Modes

- **Base addressing**
  - The processor has a special register called the *base register* or *segment register*.
  - All operand addresses generated are added to the base register to get the final memory address.
  - Allows easy movement of code and data in memory.
- **Autoincrement** and **Autodecrement**
  - First introduced in the PDP-11 computer system.
  - The register holding the operand address is automatically incremented or decremented after accessing the operand (like a++ and a-- in C).