Instruction Set Design

Chapter 13: Addressing Modes and Instruction Formats

Joannah Nanjekye

July 31, 2024

Addressing Modes

The address field or fields in a typical instruction format are relatively small. We would like to be able to reference a large range of locations in main memory or, for some systems, virtual memory.

Common addressing techniques:

- Immediate
- Direct
- Indirect
- Register
- Register Direct
- Displacement
- Stack

Notations

A = contents of an address field in the instruction.

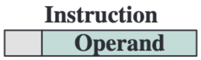
R = contents of an address field in the instruction that refers to a register.

EA = actual (effective) address of the location containing the referenced operand.

(X) = contents of memory location X or register X.

Several approaches are used, so different opcodes will use different addressing modes. Also, bits in the instruction format can be used as a mode field.

Immediate Addressing

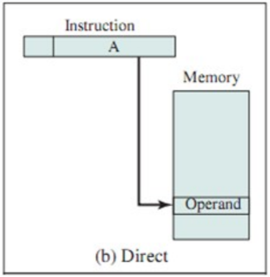


- Operand is part of the instruction
- $Operand = A$
- Example SUB 9
  - Subtract 9 from the contents of the accumulator
  - 9 is the operand
- Saves one memory or cache cycle due to no memory reference
- Affected by the limited word length (instruction sets is small compared to the word length)
- Used for constants and variable initialization

Direct Addressing

- Address field contains address of operand.

$$\boxed{EA = A}$$



(b) Direct

Example: ADD A.

- Add contents of cell A to accumulator.
- Look in memory at address A for operand.

- Single memory reference to access data.
- No additional calculations to work out effective address.
- Limited address space.

Indirect Addressing

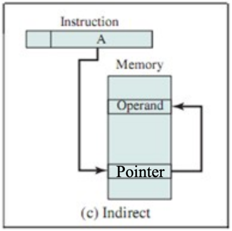
- The address field refer to the address of a word in memory.
- Memory cell pointed to by address field contains the address of (pointer to) the operand.

$$\boxed{EA = (A)}$$

  – Look in A, find address (A) and look there for operand.

Example: ADD (A).

  – Add contents of cell pointed to by contents of A to accumulator (AC).



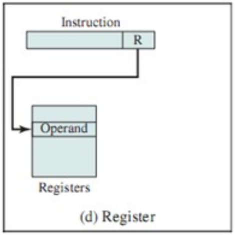
(c) Indirect

- The disadvantage is that instruction execution requires two memory references to fetch the operand (address + value).

Register Addressing

- It is similar to direct addressing. The only difference is that the address field refers to a register rather than M.M address.

$$\boxed{EA = R}$$

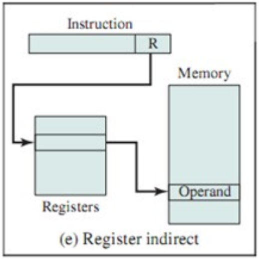


- Limited number of registers.
- No memory access.
- Very fast execution.
- The advantages are:
  - Only a small address field is needed in the instruction.
  - No time-consuming memory references are required.
- The main disadvantage is very limited address space.
- In modern Processors, multiple registers helps performance.

Indirect Register Addressing

- It is analogous to indirect addressing. The only difference is that the address field refers to a register.

$$\boxed{EA = (R)}$$



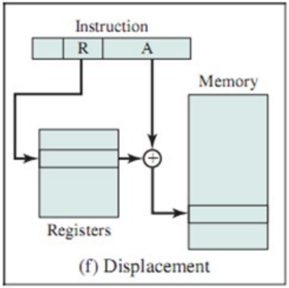
- Operand is in memory cell pointed to by contents of register R.
- Large address space (2^n).
- One fewer memory access than indirect addressing.

Displacement Addressing

"A very powerful mode of addressing that combines the capabilities of direct addressing and register indirect addressing".

$$\boxed{EA = A + (R)}$$

- The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference refers to a register (R) whose contents are added to A to produce the effective address.



Displacement Addressing ..

- There are three of the most common uses of displacement addressing:
    - Relative addressing
    - Base-register addressing
    - Indexing

(a) Relative addressing:

- R = Program counter(PC).
- EA = A + (PC).
- Get operand from A cells from current location pointed to by PC.
- This share the locality of reference & cache usage.

Displacement Addressing ...

(b) base-register addressing:

- The referenced register contains a main memory address, and the address field contains a displacement from that address.
- A holds displacement.
- R holds pointer to base address.
- The register reference may be explicit or implicit.

(c) Indexing addressing:

- The address field references a main memory address, and the referenced register contains a positive displacement from that address.
- This usage is just the opposite of the interpretation for base-register addressing.

Displacement Addressing ...

- A = base R = displacement EA = A + R
- The value A is stored in the instruction's address field, and the chosen register, called an index register, is initialized to 0. After each operation, the index register is incremented by 1.
- Some systems automatically do this as part of the same instruc tion cycle. This is known as **autoindexing**.

$$\boxed{\begin{array}{c} EA = A + (R) \\ (R) \longleftarrow (R) + 1 \end{array}}$$

- In some machines, both indirect addressing and indexing are provided, and it is possible to employ both in the same instruct ion. There are two possibilities: the indexing is performed either before or after the indirection.

Displacement Addressing ...

- If indexing is performed after the indirection, it is termed **postindexing**

$$\boxed{EA = (A) + (R)}$$

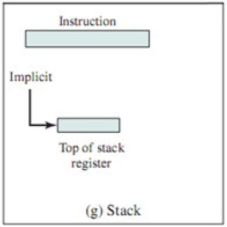
- First, the contents of the address field are used to access a me mory location containing a direct address. This address is then indexed by the register value.
- With **preindexing**, the indexing is performed before the indirection.

$$\boxed{EA = (A + (R))}$$

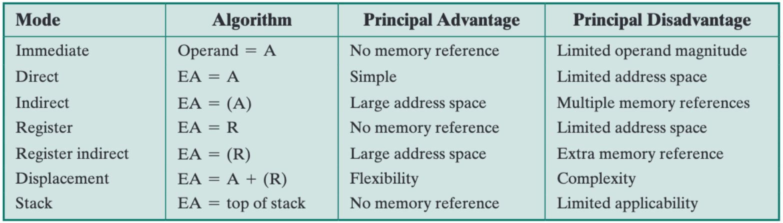
- An address is calculated as with simple indexing. In this case, however, the calculated address contains not the operand, but the address of the operand.

Stack Addressing

- A stack is a linear array of locations.
- It is sometimes referred to as a pushdown list or last-in-first-out queue.
- The top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack.
- The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.
- The machine instructions need not include a memory reference but it is (implicitly) on top of stack.



Summary



Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

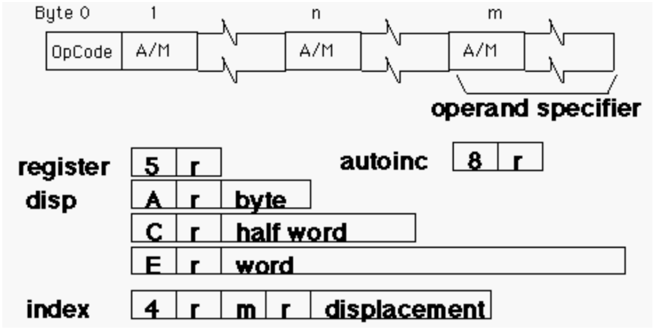
Instruction Formats

- The layout of the bits in an instruction set:
  - Opcodes
  - Operands, implicit or explicit
  - Memory address
  - Register
- Usually more than one instruction format is used and usually varies depending on whether its addressing memory, I/O devices or a register

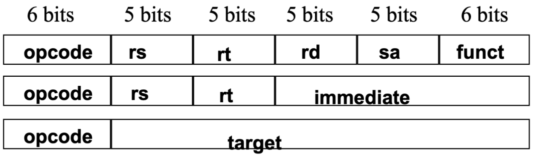
Instruction Bit Semantics

- Many instruction formats:
  - complicates decoding
  - Uses more instruction bits (to specify the format)

VAX 11 instruction format

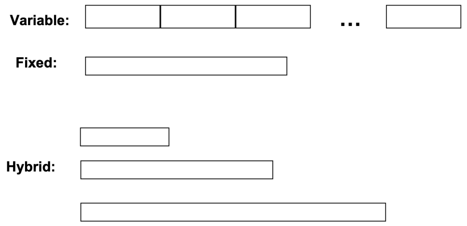


Instruction Bit Semantics



- the opcode tells the machine which format
- so add r1, r2, r3 has
  - opcode=0, funct=32, rs=2, rt=3, rd=1, sa=0
  - 000000 00010 00011 00001 00000 100000

Instruction Length



- Variable-length instructions (Intel 80x86, VAX) require multi-step fetch and decode, but allow for a much more flexible and compact instruction set
- Fixed-length instructions allow easy fetch and decode, and simplify pipelining and parallelism

Instruction Length

- Affected by and affects:
  - Memory size
  - Memory organization
  - Bus structure
  - CPU complexity
  - CPU speed

Longer Instruction Length

- Pros:
  - More opecodes
  - Operands
  - Addressing modes
  - Greater address range
  - Easier programming
- Cons:
  - Waste of space

Expanding Opecodes

- Make some opcodes short, but have a means to provide longer ones when needed
- When the opcode is short, a lot of bits are left to hold operands
- If an instruction has no operands (such as Halt), all the bits can be used for the opcode
- In between, there are longer opcodes with fewer operands as well as shorter opcodes with more operands

Example 1

- Consider a machine with 16-bit instructions and 16 registers. And we wish to encode the following instructions:
  - 15 instructions with 3 addresses
  - 14 instructions with 2 addresses
  - 31 instructions with 1 address
  - 16 instructions with 0 addresses
- Can we encode this instruction set in 16 bits?

Example 1

```
        0      0000  R1    R2    R3
        ⋮       ⋮                          15 3-address codes
        14     1110  R1    R2    R3

 0      240    1111  0000  R1    R2
 ⋮      ⋮       ⋮                          14 2-address codes
 13     253    1111  1101  R1    R2

 0      4064   1111  1110  0000  R1
 ⋮      ⋮       ⋮                          31 1-address codes
 30     4094   1111  1111  1110  R1

 0      65520  1111  1111  1111  0000
 ⋮      ⋮       ⋮                          16 0-address codes
 15     65535  1111  1111  1111  1111
```

Solution

- The first 15 instructions account for: $15x2^4x2^4x2^4 = 15x2^{12}$ = 61440 bit patterns
- The next 14 instructions account for: $14x2^4x2^4 = 15x2^8$ = 3584 bit patterns
- The next 31 instructions account for: $31x2^4$ = 496 bit patterns
- The last 16 instructions account for 16 bit patterns
- In total we need 61440 + 3584 + 496 + 16 = 65536 different bit patterns
- Having a total of 16 bits we can create 2^{16} = 65536 bit patterns

We have an exact match with no wasted patterns. So our instruction set is possible

Example 2

- Is it possible to design an expanding opcode to allow the following to be encoded with a 12-bit instruction? Assume a register operand requires 3 bits
  - 4 instructions with 3 registers
  - 255 instructions with 1 register
  - 16 instructions with 0 register

Solution

- The first 4 instructions account for: - $4x2^3x2^3x2^3 = 4x2^9 = 2048$ bit patterns
- The next 255 instructions account for: - $255x2^3$ = 2040 bit patterns
- The last 16 instructions account for 16 bit patterns
- In total we need 2048 + 2040 + 16 = 4104 bit patterns
- With 12 bit instruction we can only have 2^{12} = 4096 bit patterns

Required bit patterns (4104) is more than what we have (4096), so this instruction set is not possible with only 12 bits

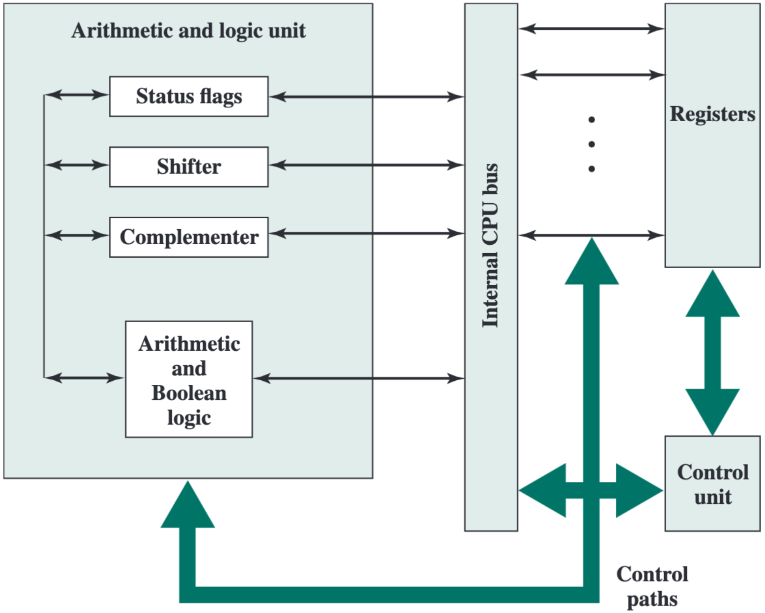
Allocation Bits

Trade-off between opcodes and the power of the addressing capability. Variable-length opecodes are helpful

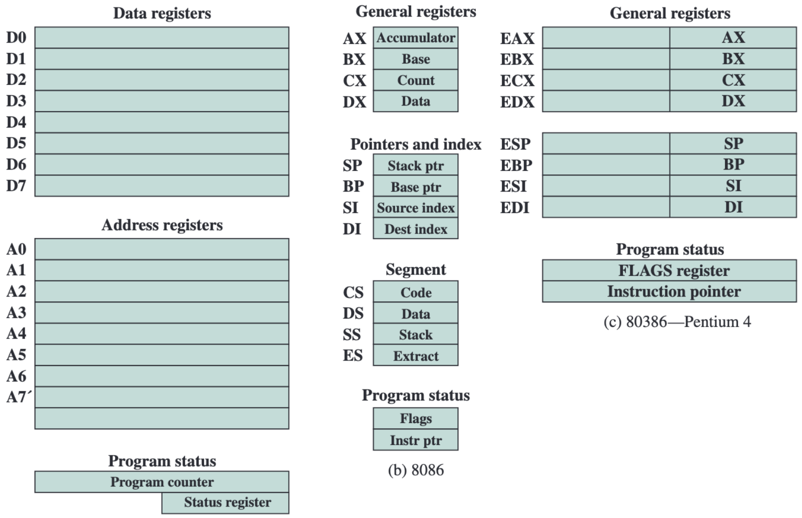
- Factors determining the use of addressing bits:
  - Number of addressing modes
  - Number of operands
  - Register vs. memory
  - Number of register sets
  - Address range

Pre-lecture 16

Processor Organization



Register Organization



(a) MC68000

(b) 8086

(c) 80386—Pentium 4

Acknowledgement of Sources

- Most of these slides are adopted from Dr. Nader Taleb lectures
- https://slideplayer.com/slide/10852334/
- https://cseweb.ucsd.edu/classes/su06/cse141/slides/s02-isa-1up.pdf
- https://slideplayer.com/slide/9163053/
- https://h3turing.vmhost.psu.edu/cmpsc312/ExpOpcodes1.pdf
- https://www.cset.oit.edu/ lynnd/cst131/ho/Lec03Slides.pdf