

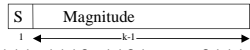
## Lecture #3

## Number Representation

- A number is a concept, for which there are many concrete representations/notations
  - e.g. the number “eleven” can be represented as 11 (decimal); XI (roman); 13 (octal); B (hex); 1011 (binary); ‘k’ (alpha); etc...
- Conversely, a single concrete representation may have several interpretations
  - e.g. the representation 10 could be interpreted as ten, eight, sixteen, two, etc...

- We have only bits available for our machine
  - binary numbers used in memory for concrete representation
  - *the corresponding data, however, can be interpreted in different ways*
    - e.g. numeric data, instruction, ASCII string, ...
- Problem: representing negative numbers in binary

## Signed Magnitude

- Use first bit to represent positive/negative
  - 
  - i.e. 1111, 1110, 1101, ..., 0111
- Q. what is the largest number?
- Q. what is the smallest number?
- Q. how many numbers total?
- Notice: for negative numbers, addition looks more like subtraction!

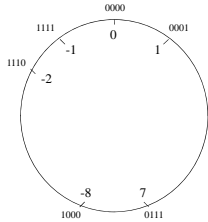
## One's Complement

- To negate a number, flip the bits
  - e.g. -5 would be:
- First bit is still the sign!
  - i.e. 1000, 1001, 1010, ..., 0110, 0111
- Now addition always “looks like” addition
- But the range is still  $2^k - 1$
- Why?

## Two's Complement

- To negate a number:
  - flip the bits
  - add 1
- e.g. -5 would be:
- Uses the full  $2^k$  range!
  - consider negating 0
- Reverse this to evaluate a (-ve) binary representation:
  - subtract 1
  - flip the bits
- e.g. 1101 would be:
- The first bit gives the sign (like signed mag)

- Imagine a circle:



- addition: CW
- subtraction: CCW
- Where does "overflow" occur?

## Number of Bits Matters

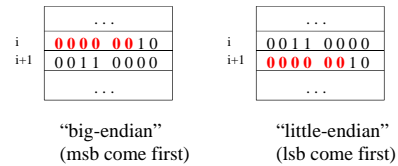
- To work with (the concrete two's complement representation of) a number, we need to know the number of bits
  - E.g., what is the value of 1101?
    - Could be -3 (4 bits), or could be 13 (8/16/... bits)
- Consider adding two numbers:
 
$$\begin{array}{r} 11111111 \\ + \quad 1101 \\ \hline \end{array}$$
  - ← is this -1 or 255?
  - ← is this -3 or 13?
  - ??????? ← should this be -4, 12, 252 or 268?
- So, arguments must be the same length

## Extending a Representation

- Two natural ways to lengthen a concrete representation:
  - zero extension
    - add 0's to the front
    - 1101 becomes 00001101 (i.e., 13)
    - 0111 becomes 00000111 (i.e., 7)
  - sign extension
    - use first bit to fill in the front
    - 1101 becomes 11111101 (i.e., -3)
    - 0111 becomes 00000111 (i.e., 7)
- Former always results in a positive number, while the latter always preserves the sign

## Endianness

- Each memory cell is only k bits
- How to represent big (ie > k bit) numbers?
  - Example: 560 (1000110000) in an 8-bit arch.



## Lecture #4

## Instructions

- Basic format of instructions in memory:
 

OP CODE	OPERAND(S)
---------	------------

  - op code → function/action
  - operand(s) → argument(s) to the function
- In general:
  - Instructions can occupy multiple cells (eg 4 bytes in x86)
  - One machine can have variable-sized instructions
- Our instructions all occupy 1 cell (16 bits)
- Operands can be interpreted in different ways...

## Addressing Modes

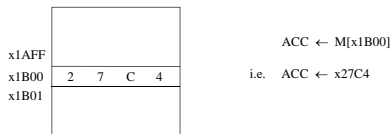
- Immediate
  - operand *is* the argument
    - LOAD #6
    - effect: ACC ← 6
- Register
  - operand gives the *register* where argument is
    - LOAD r1
    - effect: ACC ← r1

## Addressing Modes (II)

- Relative
  - commonly: PC-relative, base-relative
  - operand gives *displacement*, relative to a particular register (such as PC)
    - JMP 3
    - effect: branch forward 3 cells  
ie. PC ← PC + 3
    - (important to know what the value of PC is!)

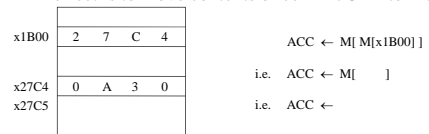
## Addressing Modes (III)

- Direct
  - operand is the *address* of the argument
    - LOAD 1B00
    - effect is to move contents of cell x1B00 into ACC



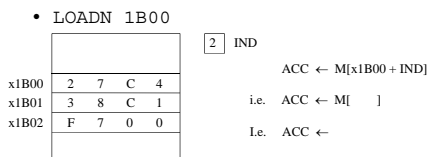
## Addressing Modes (IV)

- Indirect
  - operand is the *address of the address* of the argument
    - LOAD @1B00
    - effect is to move contents of cell x27C4 into ACC

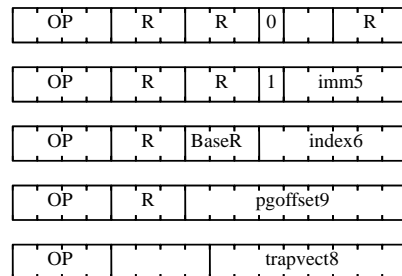


## Indexed Addressing Modes

- Indexing can be combined with other modes (e.g., direct and base-relative)
  - *address* of argument given by:  
*operand + value of a register*



## Machine Instruction Formats



- addressing modes:
  - immediate, register, base-relative, direct, indirect
- instructions can have 1, 2, or 3 operands
- examples of general syntax:
  - OP DR,SR1,SR2
  - OP DR,SR1,imm5
  - OP DR,BaseR,index6
  - OP DR,pgoffset9
  - OP trapvect8

## Categories of Instructions

- Arithmetic / Logical
  - addition, conjunction, negation
- Load / Store
  - moves data between registers and memory
- Branch
  - unconditional and conditional
  - with and without “linking” (storing PC)
- Miscellaneous
  - DEBUG prints machine state to console

## Machine Code Examples

- 0 0 0 1 0 1 0 0 1 1 1 1 1 1 1 1
- 0 1 1 0 1 0 0 0 0 0 1 0 1 1 1 1
- 0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 0
- 1111000000100101

## Machine Code Examples

- 0 0 0 1 0 1 0 0 1 1 1 1 1 1 1 1
- 0 1 1 0 1 0 0 0 0 1 0 1 1 1 1 1
- 0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 0
- 1111000000100101

## Machine Code Examples

1. ADD R2, R3, #-1  
0 0 0 1 0 1 0 0 1 1 1 1 1 1 1 1
2. LDR R4, R0, x2F  
0 1 1 0 1 0 0 0 0 0 1 0 1 1 1 1
3. BRZp x10C  
0 0 0 0 0 1 1 1 0 0 0 0 1 0 1 0
4. TRAP x25  
1111000000100101

## Effect of Instruction Execution

x502E	0101010100011000	
x502F	0011001100110001	
x5030	1111111100000001	
x5031	0000000000000000	
NZP	1 0 0	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
R0	0101000000000001	
R1	0100001100000001	
R2	1000001101100001	
R3	0111111111111111	
R4	0111111111111111	

### Effect of Instruction Execution

x502E	0101010100011000	
x502F	0011001100110001	
x5030	1111111100000001	
x5031	0000000000000000	
NZP	1 0 0	0 0 1
R0	0101000000000001	
R1	0100001100000001	
R2	1000001101100001	0111111111111110
R3	0111111111111111	
R4	0111111111111111	

### Effect of Instruction Execution

x502E		
x502F		
x5030		
x5031		
NZP		
R0		
R1		
R2		
R3		
R4		

### Effect of Instruction Execution

x502E		
x502F		
x5030		
x5031		
NZP	1 0 0	
R0		
R1		
R2		
R3		
R4	1111111100000001	

### Lecture #5

### Exercise

- Write out the memory image of a program loaded at location x3000, that:
  - computes the sum of the integers contained in the 12 locations x3100-x310B
  - prints this sum to the console

	Memory	Binary	Instruction/Data
30B0	0 0 0 4		
30B1	2 2 B 0		
30B2	E 0 B 7		
30B3	F 0 2 2		
30B4	1 2 7 F		
30B5	0 2 B 3		
30B6	F 0 2 5		
30B7	0 0 6 8		
30B8	0 0 6 9		
30B9	0 0 2 0		
30BA	0 0 0 0		

## Lecture #6

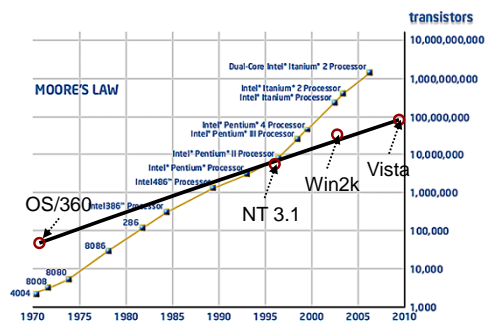
## Software Engineering

### The “Software Crisis”

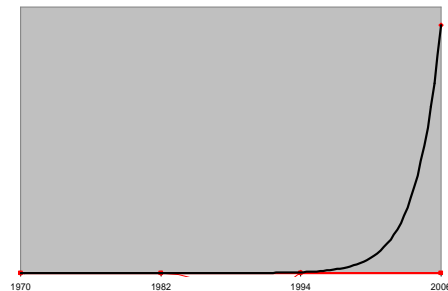
- We're in the midst of a *s/w crisis*
  - and we've been there for 30+ years!
- Complexity continuously increasing:
  - machine
  - software
- Tools and techniques to manage complexity
  - tools: CASE, analysis, testing, ...
  - techniques: languages, methodologies, ...

- However, system complexity frequently pushes the envelope...
- Net effect:
  - The support for building complex systems always seems to lag behind the systems we build (or want to build) !!

### Moore's Law

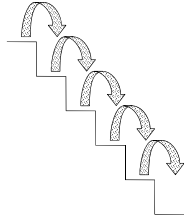


### The Great Software Crisis



## Waterfall Model of Development

- Simple model of software development
- Occurs in stages:
  1. requirements analysis
  2. system specification
  3. design
  4. implementation
  5. testing
  6. maintenance / support



## Problems with this Model

- There is no “barrier” between steps
  - e.g. begin testing *before* implementation done
- Water flows uphill
  - e.g. working on design reveals gaps in requirements analysis
  - more like an Escher print than a real waterfall!

## Fundamentals

- Many alternatives to pure waterfall exist
  - spirals, matrices, ...
- Concept of distinct *stages* is useful
  - helps structure the effort, like a “battle plan”
- Some basic stages:
  1. requirements / specification
  2. design

## Basic Stages in S/w Development: Req Analysis & System Spec

- Answers: “*What* does the system do?”
- Focus: *understanding* the problem
  - usually the software engineers are not experts in the problem domain (e.g., health care, insurance, banking...)
- Deliverables:
  1. requirements document
    - written from the user’s point of view: functionality, cost, performance
    - *defines* and *limits* the scope of the system
    - forms a contract, of sorts, between the client and the developer
  2. specification document
    - written from the developer’s point of view
    - basis for design / implementation / testing
    - document for internal consumption

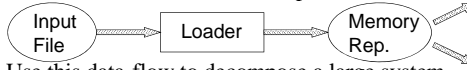
## Basic Stages in S/w Development: Design

- Answers: “*How* does the system do what it does?”
- Focus: *architecting* the software artefact that will solve the client’s problem
  - usually the client is not an expert in software engineering!
- High Level Design
  - *identify* and *evaluate* possible solutions
  - possible metrics: simplicity, effort, cost, licensing, performance,...
  - refine the design
- Lower Level Design
  - functional description of components, interfaces, and interactions
    - abstraction is critical
  - given in terms of data structures, procedures, algorithms, ...

## Basic Stages in S/w Development: Module Specification

- Answers: “What is the role of each element in the designed architecture?”
- Focus: *identifying* the abstractions
- Deliverables:
  1. Behavioural description of the modules
    - see the specification skeleton in the “Writing a Programmer’s Guide” handout
  2. Description of the interactions (interfaces)
    - see the data element dictionary section in the handout
    - other interactions include call graphs, inheritance trees, data flow, etc...
- Two common and broad classes of design:
  - procedural
  - object-oriented

## Procedural Design

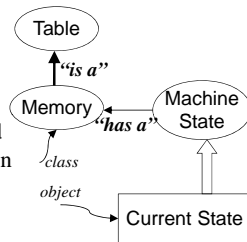
- Focus on the *functionality*
- Create a data-flow view of computation
 
- Use this data-flow to decompose a large system into individual modules
  - hierarchical: modules can then be further decomposed
- Each module responsible for some transformation
  - line of ascii text in file → parsed data in buffer
  - 4 character string of hex digits → integer

## Procedural Design (II)

- When does this process stop?
- Look for modules that:
  - are *small* enough to be easily understood
  - are *large* enough to result in reasonable overall complexity
  - are *generic* (and flexible) enough to be reused
- Key activity: **DEFINE INTERFACES**
  1. Syntactical structure
    - function name, argument types (i.e., signature)
  2. Behavioural contract
    - requirements and guarantees (i.e., requires/ensures specification)
- Fixing the interfaces allows work to proceed in parallel on the sub-parts

## Object-Oriented Design (OOD)

- Focus on the *data*
- Program = collection of interacting objects
- Sketch out the *types* needed and the *interactions* between these types



## OOD: Finding Classes

- Design is often based on reality
- Talk to field experts to understand the system being modeled in software.
- Write down scenarios
  - “use case” analysis
- Draw lots of pictures and refine model
- Decide on class invariants

## OOD: Specify Relationships

- Typical class relationships include:
  - inheritance (e.g. “a car *is a* vehicle”)
    - key principle: substitutability and polymorphism
      - a client designed to use type vehicle, can be given a car to use instead
    - therefore, all properties and behaviours of vehicles must apply to cars too
  - containment (e.g. “a car *has a* steering wheel”)
    - key principle: encapsulation
    - use (e.g. “a car *uses a* traffic light”)
- Determine responsibility of each class
  - delegate *where appropriate*
  - benefit of delegation: single point of control/modification
  - cost of delegation: can increase complexity and decrease performance
- Must strike a balance (small vs large) in the *size* and *functionality* of classes

## OOD: Specify Operations

- Important categories of operations:
  - construct, initialize, copy, assign, destroy
  - access, update, iterate
- Set should be small and independent
  - do not implement every possible use / extension
  - generality may promote reuse, sacrifice performance
- Focus on behavior, not implementation
  - confirm invariants
- Key activity: **DEFINE INTERFACES**