**What Is Computer Architecture?**

"The term architecture is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior as distinct from the organization of the dataflow and controls, the logic design, and the physical implementation."

_Gene Amdahl_, IBM Journal of R&D, April 1964

---

**Architecture, Organization, Implementation**

Computer architecture: SW/HW interface
- instruction set
- memory management and protection
- interrupts and traps
- floating-point standard (IEEE)

Organization: also called microarchitecture
- number/location of functional units
- pipeline/cache configuration

Implementation:
- logic design
- low-level circuits

---

**What Is 752 All About?**

State-of-the-art computer hardware design
Uniprocessor architecture (e.g., microprocessors)
Memory architecture
I/O architecture
Fundamentals, current systems, and future systems
752 Class Info

Instructor: Professor David Wood
 URL: http://www.cs.wisc.edu/~david
TA: Nikhil Gupta (nikhil@cs.wisc.edu)
Class info:
 URL: http://www.cs.wisc.edu/~cs752-1
Class mailing list:
 cs752-1-list@cs.wisc.edu

Who Should Take 752?

Graduate students (MS/PhD)
1. Computer Architects to be
2. Computer system designers
3. System software designers

Course expectations:
 Will loosely follow text
 Major emphasis on cutting-edge issues
 Students will read a list of research papers
 Term project

Required Background

 basic architecture (CS/ECE 552)
 basic OS (CS 537)

Other useful/related courses:
 compilers (CS 536 & 701)
 performance evaluation (CS 547 & 747)
 VLSI systems (CS/ECE 755)
 Multiprocessor architecture (CS/ECE 757)

752: Components

Texts
 Computer Architecture: A Quantitative Approach by Hennessy and Patterson, 3rd Edition
 Readings in Computer Architecture by Hill, Jouppi and Sohi

Homework
 4 to 6 homework assignments
Project
 groups of 3 or 2
Exams
 several short in-class quizzes, no advanced warning
 midterm and final exam, out of class
Grading

Grading breakdown
- Project: 30%
- Homework: 15%
- Quizzes: 10%
- Midterm: 20%
- Final: 25%

Conduct

Lectures
- Scheduled 3 per week; average 2 per week
- Consult Course Schedule for tentative schedule

Academic Misconduct
- University policy will be followed strictly

Late Policy
- Late homeworks are 50% off, unless prior arrangements are made.
- No late project will be accepted. Period.

System Architecture Hierarchy

Levels of Architecture


Level 1: System architecture
- Interface to outside world (e.g., languages, GUI)

Level 2, 3, 4: Software architecture
- 2, 3: Programming languages, and 3, 4: Operating system

Level 5: Computer architecture
- Interface between software and hardware

Level 6, 8, 9: Physical I/O

Level 7: Memory architecture
**Why Study Computer Architecture?**

**Answer #1**: Optimize cost/performance as technology changes

<table>
<thead>
<tr>
<th>Technology</th>
<th>Annual Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistor count</td>
<td>25%</td>
</tr>
<tr>
<td>Transistor speed</td>
<td>20%-25%</td>
</tr>
<tr>
<td>DRAM density</td>
<td>60%</td>
</tr>
<tr>
<td>DRAM speed</td>
<td>4%</td>
</tr>
<tr>
<td>Disk density</td>
<td>25%</td>
</tr>
<tr>
<td>Disk speed</td>
<td>4%</td>
</tr>
</tbody>
</table>

Varying rates of improvement result in different “good” solutions

**Answer #1 (Cont.)**

What do these intervals have in common?
- 1776—2000 (225 years)
- 2001—2002 (2 years)

Absolute speed improvement of computers comparable!
- If performance improves by 50%, $1.5^2 = 2.25$

**Why Study Computer Architecture?**

**Answer #2**: Innovation built into performance trends

Initially, transistor speed was primary limit to performance
- ~35% performance improvement per year

Now, larger transistor counts ➔ advanced microarchitecture
- > 50% performance improvement per year
- the added growth due to implementation/organization

**Answer #2 (Cont.)**

<table>
<thead>
<tr>
<th>1996 performance</th>
<th>Clock (MHz)</th>
<th>Performance (SPECint)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1989 projections</td>
<td>150</td>
<td>2.5</td>
</tr>
<tr>
<td>Actual</td>
<td>200</td>
<td>10</td>
</tr>
</tbody>
</table>

752 will cover:
- technologies enabling this performance growth
- technologies sustaining future growth
Why Study Computer Architecture?

Answer #3: User requirements change rapidly
Previously infeasible solutions become ubiquitous products!

- multimedia
- entertainment
- portable computing
- virtual reality/ wearable computing
- web/ network computing
- whatever you can think of....

Applications

Scientific/numeric — large memory, floating-point operations
- weather prediction, galaxy motion, nuclear explosion simulation

Commercial — very large memory, integer arithmetic, high I/O
- inventory control, e-commerce, payroll

Embedded — low power, low cost
- handheld anything, automotive, intelligent door knobs

Home computing — high data bandwidth graphics
- multimedia, games, entertainment

The Role of the Computer Architect

Define the interface
- what is above and what is below?
- what does each interface look like?

Decisions based on:
- applications
- performance
- cost
- reliability
- power
- yield
- ....

Classes of Computers

Supercomputers
- high end: ~8K processors Intel/IBM/SGI ASCI machines
- conventional: Cray T-90, T3E, X1

Commercial servers:
- large-scale servers: 64-512 processor SGI/Sun DSMs/SMPs
- medium-scale servers: 8-32 processor Sun/HP/IBM SMPs
- small-scale servers: 1-4 processor Pentium/Merced/Sun
- reliable servers: 1-20 processor IBM/Tandem

Low-cost and embedded computers:
- desktop computers: PCs and workstations
- handheld/laptop computers: laptops, wireless phones, PDAs
- special-purpose computers: game, Web TV, routers
Pipelining

- latch
- pipeline stage

Up to one result per clock cycle

- e.g., lunch buffet, automotive manufacturing, etc.
- But, performance may be limited
- unequal segments
- latch overhead
- dependences

Memory Hierarchy

- Principles of performance
  - fast common case
  - locality in accessing data and instructions
    - temporal locality — accesses to same data in near future
    - spatial locality — accesses to nearby data
- Implementation facts
  - on-chip faster than off-chip
  - SRAM faster than DRAM faster than disk
- Keep recently referenced data close to the processor
### Performance Metrics

**Time (latency)**
- Elapsed time vs processor time

**Rate (bandwidth)**
- Performance = rate = work per time

### Little's Law (-ranking)

Key Relationship between latency and bandwidth:
Average number in system = arrival rate * avg. holding time

**Example:**
How big a wine cellar should I build?
We drink (and buy) an average of 4 bottles per week
On average, I want to age my wine 5 years

bottles in cellar = 4 bottles/week * 52 weeks/year * 5 years
= 1040 bottles

---

### Memory Hierarchy (Cont.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Speed</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>&lt; 1 KB</td>
<td>1-5 ns</td>
<td>9,600 MB/s</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>&lt; 256 KB</td>
<td>10 ns</td>
<td>3,200 MB/s</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>&lt; 8 MB</td>
<td>30 ns</td>
<td>800 MB/s</td>
</tr>
<tr>
<td>Memory</td>
<td>&lt; 4 GB</td>
<td>100 ns</td>
<td>133 MB/s</td>
</tr>
<tr>
<td>Disk</td>
<td>&gt; 1 GB</td>
<td>20 ms</td>
<td>4 MB/s</td>
</tr>
</tbody>
</table>

- **Performance & Cost**
  - Performance metrics
  - Elapsed time vs CPU time
  - Iron law of performance
  - Benchmarks and benchmarking
  - Averaging
  - Amdahl’s law
  - Balance and bursty behavior
  - Cost and price

- **Performance Metrics**
  - Time (latency)
  - Rate (bandwidth)

- **Little’s Law (-ranking)**
  - Key Relationship between latency and bandwidth:
  - Average number in system = arrival rate * avg. holding time

- **Example:**
  - How big a wine cellar should I build?
  - We drink (and buy) an average of 4 bottles per week
  - On average, I want to age my wine 5 years

  bottles in cellar = 4 bottles/week * 52 weeks/year * 5 years
  = 1040 bottles
**MIPS (مخالطة)***

MIPS = instruction count/(execution time x 10^6)
= clock rate/(CPI x 10^6)

Need a uniform measure of work
- instruction sets are not equivalent
- different programs use a different instruction mix
- instruction count is not a reliable indicator of work
- some optimizations add instructions
- instructions have varying work

---

**Relative MIPS (مخالطة)***

Relative MIPS = (time\textsubscript{reference}/time\textsubscript{new}) x MIPS\textsubscript{reference}
Relative MIPS (e.g., Pentium MIPS) a bit better

But, very sensitive to reference machine!
How do compilers and OS compare?
Bottom line: maybe useful if same instruction set/compiler/OS

---

**MFLOPS (مخالطة)***

MFLOPS = (FP ops/program) x (program/time) x 10^6

Not great because it
- ignores other instructions (e.g., load/store)
- depends on how FP-intensive program is
- many programs have no FP ops
- others are dominated by FP ops

Beware of peak MFLOPS!

---

**Normalized MFLOPS (مخالطة)***

Normalized FP: give canonical # FP ops to prog
Normalized MFLOPS = (# canonical FP ops/time) x 10^6

Not all machines have the same FP ops
- Cray traditionally does not implement divide
- Motorola has SQRT, SIN, and COS
Not all FP ops are same work
- adds usually faster than divide
Iron Law (✝)

Time/prog = insts/prog x cycles/inst x sec/cycle
sec/cycle (a.k.a. cycle time, clock time)
  ▫ mostly determined by technology and CPU organization
cycles/inst (a.k.a. CPI)
  ▫ mostly determined by ISA and CPU organization
  ▫ overlap among instructions makes this smaller
insts/prog (a.k.a. instruction count)
  ▫ instructions executed NOT static code
  ▫ mostly determined by program, compiler, ISA

Some have argued:
▫ CISC CPU time = P x 8 x T = 8PT
▫ RISC CPU time = 2P x 2 x T = 4 PT
▫ RISC CPU time = CISC CPU time/2

The truth is much more complex!

Performance Comparison

Machine A is n times faster than machine B iff
▫ perf(A)/perf(B) = time(B)/time(A) = n
▫ Also called speedup of A over B
Machine A is x% faster than machine B iff
▫ perf(A)/perf(B) = time(B)/time(A) = 1 + x/100

E.g., Suppose A takes 10s, B takes 15s
▫ 15/10 = 1.5 ➔ A is 1.5 times faster than B
▫ 15/10 = 1 + 50/100 ➔ A is 50% faster than B

Simple Example

<table>
<thead>
<tr>
<th>Op</th>
<th>Frequency</th>
<th>Cycle count</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>43%</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>21%</td>
<td>1</td>
</tr>
<tr>
<td>Store</td>
<td>12%</td>
<td>2</td>
</tr>
<tr>
<td>Branch</td>
<td>24%</td>
<td>2</td>
</tr>
</tbody>
</table>

Assume stores can execute in 1 cycle by slowing clock 15%
Should this be implemented?
Simple Example (Cont.)

Old CPI = 0.43 + 0.21 + 0.12 x 2 + 0.24 x 2 = 1.36
New CPI = 0.43 + 0.21 + 0.12 + 0.24 x 2 = 1.24

Speedup = old time/new time
\[ = \frac{(P \times \text{old CPI} \times T)}{(P \times \text{new CPI} \times 1.15T)} \]
\[ = \frac{1.36(1.24 \times 1.15)}{0.97} \]

Answer: don’t make the change!

Type of Benchmarks

Real programs
- representative of real workload
- only accurate way to characterize performance
- requires considerable work

Kernels
- “representative” program fragments
- good for focusing on individual features not big picture

Mixes
- instruction frequency of occurrence; calculate

Other Benchmarks

Toy benchmarks
- e.g., fibonacci, prime number
- little number

Synthetic benchmarks
- programs intended to give specific mix
- ignore dependencies
- maybe ok for non-pipelined, non-cached, w/o optimizing compilers

SPEC2000 Benchmarks

Integer
- gzip compression using Lempel-Ziv
- vpr FPGA placement and routing
- gcc GNU C compiler
- mcf optimization of public transit scheduling
- crafty chess-playing program
- parser syntactic English language parser
- eon graphics visualization
- perlmbk Perl with 4 input scripts
- gap group theory application package
- vortex object-oriented database
- bzip2 block-sorting compression algorithm
- twolf VLSI place and route
**SPEC2000 Benchmarks**

- Floating point
  - wupwise
  - swim
  - mgrid
  - apply
  - mesa
  - gaigel
  - art
  - equake
  - facevec
  - ammp
  - lucas

**SPEC2000 benchmarks, contd.**

- Floating point, contd.
  - fma3d
  - sixtrack
  - apsi

---

**SPEC Evolution**

- SPEC89: 10 programs (4 integer, 6 FP)
- SPEC92: 15 programs (5 integer, 10 FP)
- SPEC95: 18 programs (8 integer, 10 FP)
- SPEC2000: 25 programs (11 integer, 14 FP)

Caveat: Only useful if they match your workload!

- e.g., I completely ignore SPECFP and focus on gcc
gcc behaves most similar to our simulator

---

**Kernel Example**

- Inner product
  - DO L = 1, LP
  - Q = 0.0
  - DO K = 1, N
  - \( Q = Q + Z(K)^*X(K) \)
**Synthetic Benchmark Example**

Dhrystone, Whetstone

\[
\begin{align*}
X &= 1.0 \\
Y &= 1.0 \\
Z &= 1.0 \\
\text{DO } I &= 1, N8 \\
\text{CALL } P3(X,Y,Z)
\end{align*}
\]

SUBROUTINE P3(X,Y,Z)

\[
\begin{align*}
X_1 &= X \\
Y_1 &= Y \\
X_1 &= T \cdot (X_1 - Y_1) \\
Y_1 &= T \cdot (X_1 + Y_1) \\
Z &= (X_1 + Y_1)/T^2 \\
\text{RETURN}
\end{align*}
\]

**Mix Example**

Gibson Mix, developed in 1950's at IBM

- load/store 31%
- branch 17%
- fixed add/sub 6%
- float add/sub 7%
- fixed div 2%
- float div < 1%
- logical 2%

Generally speaking, these numbers are still valid today

**SPEC Benchmarking Process**

Steps:
1. For each benchmark i, look up \( T_{\text{base},i} \)
2. For each benchmark i, run target machine to get \( T_{\text{new},i} \)
3. Compute geometric mean: \( T_{\text{mean}} = \left( \prod_{i=1}^{n} T_{\text{base},i}/T_{\text{new},i} \right)^{1/n} \)

But,
- What workload does this match?
- How does geometric mean predict performance?
**SPEC Benchmarking Process**

Steps:
- extract benchmarks from applications
- choose performance metric
- execute benchmarks on candidate machines
- project performance in new machine

**How to Average**

Arithmetic mean
- sum of all execution times/number
- valid only if all programs run equally often

Weighted arithmetic mean
- "weight" factors to give equal importance to all programs

**Other Averages**

E.g., 30 mph for first 10 miles, 90 mph for next 10 miles
What is the average speed?

Average speed = (30 + 90)/2 = 60 mph

Average speed = total distance / total time
= 20 / (10/30 + 10/90)
= 45 mph

**Harmonic Mean**

Harmonic mean of rates = 1/(Σ(1/rate(i))/n)

Use harmonic mean if forced to start and end with rates
key insight: harmonic mean converts rates to times,
computes arithmetic mean of times, then
converts back to rates.
Dealing with Ratios

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine A</td>
<td>Machine B</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>1000</td>
</tr>
</tbody>
</table>

If we take ratios with respect to Machine A (Machine B)

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine A</td>
<td>Machine B</td>
</tr>
<tr>
<td></td>
<td>1 (0.1)</td>
</tr>
<tr>
<td></td>
<td>10 (1)</td>
</tr>
<tr>
<td></td>
<td>1 (10)</td>
</tr>
<tr>
<td></td>
<td>0.1 (1)</td>
</tr>
</tbody>
</table>

Averages different depending on which is the reference

Geometric mean of ratios = \( \prod_{i=1}^{n} \frac{1}{\text{ratio}(i)} \)

Use geometric mean if forced to use ratios
Independent of reference machine
In the example, mean for machine A and B is 1
 normalized with respect to either machine

But.....

Geometric mean of ratios is not proportional to total time
Arithmetic mean in example says machine B is 9.1 times faster
Geometric mean says they are equal
If we took total execution time, A an B are equal only if
program 1 is run 100 times more than program 2
Geometric mean will mispredict for three or more machines
Use AM for times, HM for rates, GM for ratios

Pitfalls

Choosing performance from the wrong application space
 e.g., in a realtime environment, choosing troff
Choosing benchmarks from no application space
 e.g., synthetic workloads
Using toy benchmarks
 e.g., used to prove the value of RISC in early 80's
Mismatch of benchmark properties with scale of features studied
 e.g., using SPEC for large cache studies
Pitfalls...

Carelessly scaling benchmarks
- Truncating benchmarks
- Using only first few million instructions
- Reducing program data size

Too many easy cases
- May not show value of a feature

Too few easy cases
- May exaggerate importance of a feature

Amdahl’s Law

Why you shouldn’t ignore the Uncommon case?

\[
\text{Speedup} = \frac{\text{new time}}{\text{old time}} = \frac{1}{1 - f + f/s}
\]

Let an optimization speed \( f \) fraction of time by a factor of \( s \)

\[
\text{Speedup} = \frac{(1 - f) + f * oldtime}{(1 - f) * oldtime + f/s * oldtime} = \frac{1}{1 - f + f/s}
\]

\( f = 95\% \) and \( s = 1.10 \) (speedup common case)

\[
\text{Speedup} = \frac{1}{(1 - 0.95) + 0.95/1.10} = 1.094
\]

\( f = 5\% \) and \( s = 10 \)

\[
\text{Speedup} = \frac{1}{(1 - 0.05) + 0.05/10} = 1.047
\]

\( f = 5\% \) and \( s \to \infty \)

\[
\text{Speedup} = \frac{1}{(1 - 0.05) + 0.05/\infty} = 1.052
\]

The example illustrates that common case should be sped up

Amdahl’s law

\[
\lim_{S \to \infty} \frac{1}{1 - f + f/s} = \frac{1}{1 - f} \rightarrow \text{Make common case fast}
\]
Amdahl was talking about a parallel processor with large speedup. At some point you have to pay attention to the serial part.

At a system level, bandwidths and capacities should be balanced. Each level capable of demanding / supplying bandwidths. Refer to memory hierarchy figure.

**Amdahl's law**

![Graph](image)

At some point you have to pay attention to the serial part.

**Balance**

At a system level, bandwidths and capacities should be balanced. Each level capable of demanding / supplying bandwidths. Refer to memory hierarchy figure.

**Balance: Example**

\[ \text{IPC} = 1.5 \left( \frac{1}{\text{CPI}} \right) \]

- 0.30 loads and stores
- 0.90 data cache hit rate
- 0.95 icache hit rate
All cache misses require 32 bytes

So processor demand is:

\[ 1.5 \times 0.3 \times 0.10 \times 32 \text{ (data) } + \]
\[ 1.5 \times 1.0 \times 0.05 \times 32 \text{ (instructions) } \]
\[ = 3.8 \text{ bytes/clock} \]
(to keep processor busy memory needs to supply)

**Balance...**

If demand bandwidth = supply bandwidth of a resource

- computation is resource-bound

If memory bandwidth = processor demand for program P

- P is said to be memory-bound

Same for CPU-bound, disk-bound or I/O bound

**Goal: to be bound everywhere (Why?)**
Memory Bandwidth

- copy: \( a[i] = b[i] \)
- scale: \( a[i] = q * b[i] \)
- sum: \( a[i] = b[i] + c[i] \)
- saxpy: \( a[i] = b[i] + q * c[i] \)

![Graph showing memory bandwidth](image)

### Memory Bandwidth...

Memory bandwidth of real systems (MB/s)

<table>
<thead>
<tr>
<th>System</th>
<th>copy</th>
<th>scale</th>
<th>sum</th>
<th>saxpy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray C90</td>
<td>7000</td>
<td>7000</td>
<td>9400</td>
<td>9500</td>
</tr>
<tr>
<td>Cray T3D</td>
<td>19800</td>
<td>10200</td>
<td>13000</td>
<td>13700</td>
</tr>
<tr>
<td>Alpha 150MHz</td>
<td>98</td>
<td>90</td>
<td>68</td>
<td>90</td>
</tr>
<tr>
<td>Cray T3D</td>
<td>380</td>
<td>330</td>
<td>190</td>
<td>180</td>
</tr>
</tbody>
</table>

### Balance (again)

Storage capacity and bandwidth requirements
- e.g., large cache \(\Rightarrow\) higher hit rate \(\Rightarrow\) lower demand
- or, large memory \(\Rightarrow\) less paging \(\Rightarrow\) lower I/O demand

Amdahl’s rule:
- \(1 \text{ MIPS} \leftrightarrow 1 \text{ MB memory} \leftrightarrow 1 \text{ Mbits/s I/O}\)
- if corrected to 1 Mbytes/s of I/O, the rule is still good!

### Bursty Behavior

To get 2 IPC how many instructions should you —
- fetch per cycle?
- issue per cycle?
- commit per cycle?
- the answer is NOT 2

Instructions are not like sand where peaks and valleys are leveled
Bursty Behavior

Example,
\[ a = b + c \]
\[ d = e \times f \]
Dependences will cause pipeline stalls (or bubbles or wait times)
So sometimes pipeline will be full and at other only partially full,
\( \Rightarrow \) a higher peak level is needed for a desired average level performance

Cost

Cost is very important to most real designs
Cost changes over time
\( \Rightarrow \) Learning curve lowers manufacturing costs
\( \Rightarrow \) Technology improvements lower costs e.g., DRAM

IC Cost

Cost (IC) = \( \frac{\text{Cost (die)} + \text{Cost (testing)} + \text{Cost (packaging)}}{\text{yield}} \)
\[ \text{Cost (die)} = \frac{\text{Cost (wafer)}}{(\text{die/wafer}) \times \text{yield (die)}} \]
\[ \text{Yield (die)} = \text{yield (die)}} \times 1 + \left( \frac{\text{defects/cm}^2 \times \text{area}}{\alpha } \right)^{-\alpha} \]
Often, \( \alpha \) is 0.30
Cost (die) = \( f \) (die area)

Cost Breakdown

Component Cost
\( \Rightarrow \) Microprocessor, SRAM, DRAM + disk
\( \Rightarrow \) Power supplies, packaging
Direct costs
\( \Rightarrow \) Manufacturing (labor, scrap) + warranty
Indirect costs
\( \Rightarrow \) R & D + marketing
\( \Rightarrow \) administrative
\( \Rightarrow \) profit + taxes
Price

Only loosely related to cost!

Component Cost +
25 to 40% direct cost +
45 to 65% gross margin = Average Selling Price

Add 60 to 75% to correct discounts and allow dealer profits = List Price

Note:
❖ component cost: 15 to 30%
❖ R&D: 8 to 15% of list price

BGvN46: Concepts

Classic paper
❖ most observations are still true
❖ most historians credit Eckert and Mauchly for this idea

“It is evident that the machine must be capable of storing in some manner not only the data but also the instructions which govern the actual machine”

“conceptually we have discussed above two different forms of memory: storage of numbers and storage of orders the memory organ can be used to store both numbers and orders.”

BGvN46: Arithmetic

Binary arithmetic
Two’s complement
Iterative carry
Iterative multiply (carry-save adders)
Rounding vs. Jamming
Non-restoring division
No floating-point. Why?

BGvN46: control

40-bit data
20-bit instructions
❖ 8-bit opcode
❖ 12-bit addresses
Basic instructions
❖ conditional and unconditional branches
❖ data transfer
❖ ALU and shift
❖ Store into order – Why?