Computer Architecture

Adrian Crăciun

January 9, 2018

Contents

List of Figures

Lecture Organization

Organizational Items

- Computer Architecture:
  - how computers do what they can do
  - or, where do your programs go when you "launch" them (and how)?
- Lecture sources:
  - Based on Andrew S. Tanenbaum, *Structured Computer Organization*, 6th Edition. (Romanian translation available for 4th Edition), [Tanenbaum, 2005] and John L. Hennessy, David A. Patterson, Compute Architecture A Quantitative Approach, [Hennessy and Patterson, 2012].
  - Many figures in these notes/slides are taken from the above, exceptions will be indicated.
  - Other materials were consulted in the elaborations of these notes: e.g. [East, 1990], [Harris and Harris, 2007], [Hsu, 2001].
  -
- Exercises/Lab: presence is compulsory (University policy). For lectures 50% (but you really want to do 100%).
- Evaluation:
  - Exam + exercises + presentations + projects + involvement (weights to be discussed).
  - What if you cannot show up for labs (e.g because you work): CONTACT ME ASAP!!!
- Other items?

1 Computer Architecture - Overview and Motivation

1.1 The Structured Organization of Computers

Computers, Programs

- Computer = machine that can solve problems by carrying out instructions given to it.
- Program = a sequence of instructions describing how a certain task is performed.
- Computers are built from electronic circuits. Only very basic instructions can be carried out on such machines:

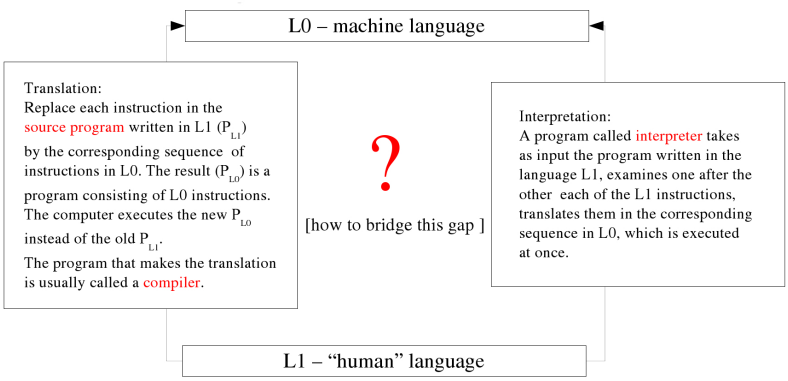


Figure 1: Moving between language levels.

  - add 2 numbers,
  - check to see whether a number is 0,
  - copy a piece of data from one location to another.

- Machine language = the set of instructions that can be carried out on the electronic circuits that form a computer.
- People find it extremely difficult to understand such a language.
- People must understand the language in order to write programs.
- To bridge the gap between machine language and human (programmer's) language, a series of abstractions have to be employed, leading to the structured organization of computers.

Translation and interpretation

Figure 1 presents the problem of moving between "machine" and "human" languages, and its solution.

Comparison: translation and interpretation

- Similar:
  - instructions from L1 are ultimately carried out by executing equivalent sequences in L0.
- Different:
  - Translation:
    * a L1 program is converted into a L0 program;
    * the L1 program is thrown away;

    - the L0 program is loaded into the memory and executed;
    - the new L0 program has control of the execution.
  - Interpretation:
    - after each L1 instruction is analyzed and decoded, it is carried out immediately (no translated program);
    - the interpretor is in control of the program, the initial L1 program is just data.

Virtual machines

- Instead of thinking in terms of interpretation and translation, imagine a hypothetical computer, a virtual machine M1, able to "speak" the language L1.
- If the machine M1 can be constructed easily, there is no need for L0 (and the corresponding machine M0).
- If M1 is too difficult to construct, one can still write programs in L1 (for M1), and these can either be translated or interpreted into L0 (and run on M0).
- However, in order to make the translation/interpretation from/of L1 to/into L0, these language should not be "too different". The initial picture ("human" - "machine") was too optimistic.
- Obvious solution: introduce intermediary levels (and virtual machines), so that moving between levels becomes easier. This leads to a hierarchy of language layers.

Multilevel machines

Figure 2 presents the structure of a multilevel machine.

Languages, virtual machines

- A virtual machine defines its machine language (as the set of all instructions which can be executed on that machine).
- A language defines its machine (as the machine that can execute all programs written in the respective language).
- Machines based on arbitrary languages can be arbitrarily hard to build (complicated, expensive).
- Example: C++, Cobol machines can be constructed with the technology today, but would not be cost effectives.
- A computer with n levels can be seen as n different machines, each with its own language.
- The terms "level" and "virtual machine" can be used interchangeably.

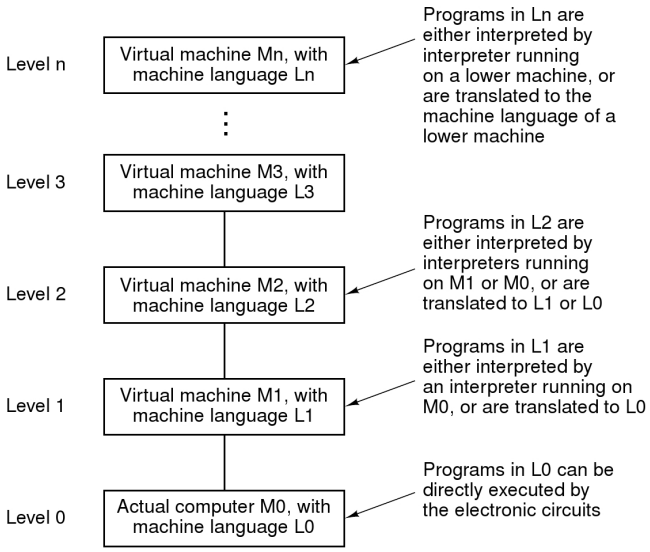


Figure 2: A multilevel machine.

Modern multilevel machines.

- Contemporary multilevel machines have two or more levels. Figure 9 presents a machine with 6 levels

The digital logic level

- Gates:
  - built from analog devices, can be accurately modeled as digital devices,
  - each gate has one or more digital inputs (i.e. 0,1) and generate simple digital outputs,
  - each gate can be built out of a handful of transistors,
  - a small number of gates can be combined to form 1 bit memories (stores for one of two values: 0,1).
- Registers:
  - combinations of typically 16, 32, 64 1 bit memories,
  - each can hold a simple binary number up to some maximum.
- Gates can also be combined to form the main computing engine.

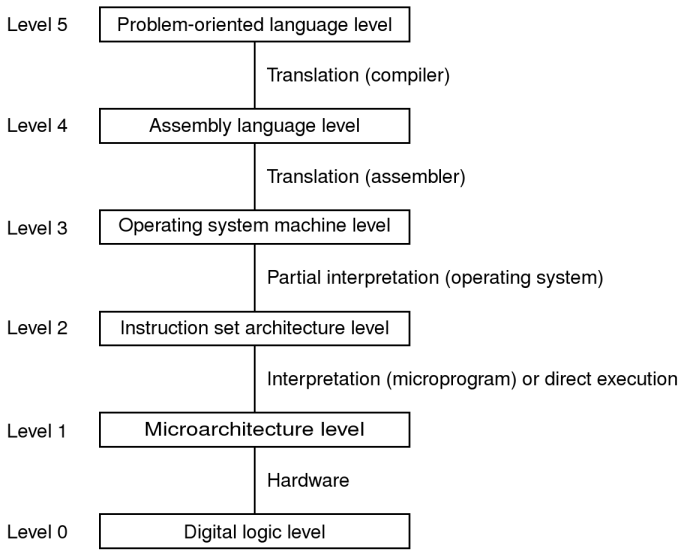


Figure 3: A multilevel machine with 6 levels. The way of moving between levels (translation/interpretation) is indicated along with the name of the program to do this.

The microarchitecture level

- A collection of (typically) 8 to 32 registers that form a local memory.
- A circuit – arithmetic logical unit (ALU) – capable of carrying out simple operations.
- Data path that connects the registers to the ALU, ensuring the data flow.

  Example: select 2 registers, pass the content to the ALU where the values are added, then store the result in some other register.
- On some machines, the operations in the data path is controlled by a microprogram, in others this is done directly in hardware.

The instruction set architecture (ISA) level

- This is what is usually described in the "Machine language reference manual".
- It contains the machine's instruction set.
- These instructions are carried out interpretatively by the microprogram or by the hardware execution circuits.

The operating system machine level

- Most instructions in the language of this level can be found also on the ISA level.
- However, in addition:
  - there is a new set of instructions,
  - the memory is organized in a different way,
  - it is possible to run two or more programs in the same time.
- The new instructions are interpreted by an interpretor at the ISA level (historically called the operating system).
- The instructions that are identical to those at the ISA level are interpreted by the microprogram or the hardware (at the microarchitecture level).
- Note that the operating system machine level is a hybrid level (instructions interpreted at 2 levels).

Some considerations regarding the levels

- The levels discussed so far:
  - are intended primarily for running interpretors and translators to support higher levels;
  - these are written by system programmers.
- The remainding levels are for application programmers.
- Other differences:
  - method of language support:

    levels 1-2-3: interpretation,
    levels 4-5: (mostly) translation;
  - nature of the language:

    levels 1-2-3: numeric (hard to read by humans),
    levels 4-5: words and abbreviations.

The assembly language level

- The language presents an abbreviation for the languages at levels 1-2-3.
- Programs in assembly languages are first translated, then interpreted by the appropriate machine for which instructions were intended.
- The program that performs the translation is called the assembler.

The problem oriented language level

- Contains the languages for application programmers (high level languages).
- There are hundreds of such languages.

  Basic, C, C++, Java, LISP, Prolog, Mathematica, Python, ... (add your favorite).

- Generally, the programs written in one of the high level languages are (can be) translated into level 3 or level 4 by translators called compilers.

Hardware, software - some considerations

- Initially:

  HARDWARE: the electronic circuits (level 0) along with the memory and the input-output devices.

  SOFTWARE: algorithms (detailed instructions telling how to do something) and their computer representations (programs).

- However, in time, the distinction between hardware and software blurred:

  "Hardware and software are logically equivalent".

  "Hardware is petrified software".

  – Any operation performed by software can be built into the hardware.

  – Any hardware operation can be simulated by software.

- The above points will be illustrated by taking a look at the development of multilevel machines.

Invention of microprogramming

- In the 1940s computers had two levels: digital logic and instruction set architecture:

  – they were complicated to build,

  – difficult to understand,

  – unreliable.

- 1951, Maurice Wilkes (University of Cambridge) had the idea to introduce an additional level (microarchitecture):

  – this meant simplified hardware,

  – a built-in interpreter at the microarchitecture level was interpreting programs at the ISA level.

Invention of the operating system

- Even with microprogramming, working with computers was tedious:
  - programs were stored on punch cards (e.g. about 80 for a program),
  - the user would have to load the interpreter/translator,
  - then load their own program and data.
- Work time had to be booked in advance – computers were complicated machines that had to be operated directly by the programmer.
- Around 1960 the idea of having a program (the operating system) in the computer at all times emerged:
  - it led to a new virtual machine that got more and more complicated over the years,
  - some of its instructions were similar to those of the ISA level,
  - others (in particular I/O) were different (operating system macros, supervisor calls).

Migration of functionality to microcode

- In the 1970s microprogramming was widespread.
- It was common practice to add more and more instructions to the set of machine instructions.
- This led to an explosion of machine instruction sets.
- Examples of instructions added:
  - integer multiplication and division,
  - floating-point arithmetic,
  - calling and returning from procedures,
  - speeding up loops,
  - handling character strings.
- Examples of added features:
  - computations involving arrays,
  - memory relocation facilities,
  - interrupt system,
  - process switching.
- Adding many instructions (not always needed, not always executed) meant that microprograms grew slower, bulkier.

Elimination of microprogramming

- The solution to the problems caused by the large instruction sets developed in the golden years of microprogramming was to reduce vastly the instruction set and have the instructions executed directly by hardware, rather than by a microprogram.
- There are two trends:
  - CISC (Complex Instruction Set Computers),
  - RISC (Reduced Instruction Set Computers),
  - more details discussed later ...

To summarize...

The evolution of multilevel machines shows how, depending on various factors, the border between hardware and software moves back and forth.

1.2 Milestones in Computer Architecture

Generations of computers

- Generation zero: mechanical computers (1623(?)–1945).
- The first generation: vacuum tubes (1945–1955).
- The second generation: transistors (1955–1965).
- The third generation: integrated circuits (1965–1980).
- The forth generation: very large scale integration (1980 – ?).
- The fifth generation: Japan's fifth generation computer project (1982 – 1992).
- The next generation?

Generation zero: mechanical computers

- Wilhelm Schichard (1623) - mechanical device able to do addition, multiplication, division. *http://en.wikipedia.org/wiki/Wilhelm_Schickard*
- Blaise Pascal (1623–1662) - mechanical machine able to do addition, substraction. *http://en.wikipedia.org/wiki/Pascal's_calculator*.
- Gottfired Wilhelm "Calculemus!" von Leibniz (1646 –1716) - mechanical machine that could multiply and divide. *http://en.wikipedia.org/wiki/Stepped_Reckoner*
- Charles Babbage (1792 – 1871):
  - difference engine:

    - designed to computer tables of numbers for navigation,
    - one algorithm: the method of finite differences using polynomials,
    - output: punched on a copper plate (CD ROM principle).
    - *http://en.wikipedia.org/wiki/Difference_engine*
  - analytical engine (1834)
    - 17000 sterling pounds from the government + a large part of family fortune,
    - machine consisting of a store (memory) of 1000 words, each of 50 decimals, the mill (computational unit – addition, substraction, multiplication, division), input section (punched cards), output section (punched cards, printed).
    - *http://en.wikipedia.org/wiki/Analytical_engine*
    - general purpose device,
    - programmable (in a simple assembly language)
    - Lady Ada Augusta Lovelace (daughter of Lord Byron) – the world's first programmer,
    - however, the analytical engine was difficult to implement, with many hardware bugs (thousands of components, entirely mechanical).
  -

- Konrad Zuse (late 1930's, Germany)
  - builds a series of automatic calculating machines using electromagnetic relays,
  - was not granted government funding (other priorities),
  - the machine was destroyed by Allied bombing of Berlin (1944), no influence on subsequent development of computers,
  - *http://en.wikipedia.org/wiki/Konrad_Zuse*
- John Atanasoff - Iowa State College (late 1930's)
  - tries to build a machine for binary arithmetic,
  - using capacitors for memory (similar to RAM chips),
  - his vision was not supported by the technology, and the machine was never completed,
  - *http://en.wikipedia.org/wiki/John_Vincent_Atanasoff*
- George Stibbitz - Bell Labs (late 1930's): implements a calculator simpler than Atanasoff's, but which worked. (*http://en.wikipedia.org/wiki/George_Stibitz*)
- Howard Aiken

  - was completing a PhD at Harvard,
  - he discovered Babbage's work and set out to implement his analytical engine using relays instead of toothed wheels,
  - Mark I, Harvard 1944 had 72 words of 23 decimal digits each, an instruction time of 6 sec, I/O on punched paper.
  - *http://en.wikipedia.org/wiki/Harvard_Mark_I*

Vacuum tubes (1945-1955)

- The development of these machines was driven by World War II.
- Colossus (1943) - Alan Turing, others:
  - built to perform the huge computations needed to break the German Enigma code,
  - secret for 30 years, no influence on the further development of computers,
  - *http://en.wikipedia.org/wiki/Colossus_computer*.
- In 1943 John Mauchly is awarded a government grant to build a machine to compute artillery tables .
  - together with J. Presper Eckert: ENIAC (Electronic Numerical Integrator and Computer), 1946.
  - 18000 vacuum tubes, 5000 relays, 30 tons, 20 registers each holding 10 digit decimal numbers, programmed by 6000 switches and connecting sockets,
  - presented at a summer school, leading to an explosion of interest in computers.
  - spinoffs: EDSAC (University of Cambridge, Maurice Wilkes), JOHNIAC (Rand Corporation), ILLIAC (University of Illinois), MANIAC (Los Alamos Labs), WEIZAC (Weizmann Institute, Israel).
- John von Neumann - Princeton Institute of Advanced studies:
  - notices that programming with switches is tedious,
  - programs can be stored along with the data in the memory,
  - binary arithmetic is better than decimal.
- The design known as the von Neumann machine is the basis of nearly all digital computers (up to this day) see Figure 4

  .
- *http://en.wikipedia.org/wiki/Von_Neumann_architecture*

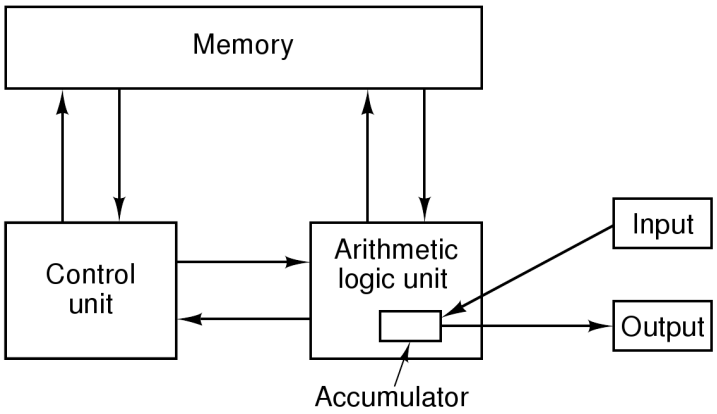


Figure 4: The von Neumann design.

- The design was implemented as the IAS machine:
  - memory of 4096 words, each of 40 bits (holding either 2 20 bit instructions - 8 bit instruction type, 12 memory address).
  - ALU with a 40 bit register - the accumulator,
  - typical instructions: add a word from memory to the content of the accumulator, write the content of the accumulator into the memory.

- Whirlwind I, a machine designed for real-time control (as opposed to number crunching like IAS, ENIAC).

  16 bit words,
  - it lead to the invention of the magnetic core memory (Jay Forrester).
- IBM - a producer of punchers and card sorting machines started showing interest in computing machines.
  - 1953 IBM 701, 2084 36 bit words,
  - 1957 IBM 704 4k core memory, 36 bit instructions, floating point hardware,
  - IBM 709, an upgrade of the 704.

Transistors (1955-1965)

- Transistors:

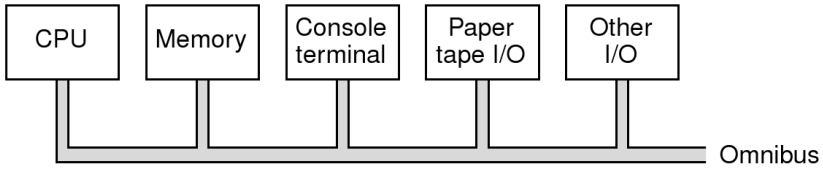


Figure 5: The PDP-8 omnibus.

  - 1948, AT&T Bell Labs, John Bardeen, Walter Brattain, William Shockley.
  - 1965 Nobel Prize for Physics,

- First transistor computers developed at MIT:
  - TX-0 (Transistorized eXperimental computer 0) 1956
  - TX-2, 1958

- From MIT, a number of teammembers go on to establish Digital Equipment corporation:
  - 1961: PDP-1:
    * 4 kb of memory, 18 bit words, $ 120,000.
    * 512 x 512 points screen,
    * Spacewar - one of the earliest games (MIT students).
    * *http://en.wikipedia.org/wiki/PDP-1*.
    * Note that at the same time, IBM's 7090, dedicated to scientific computing was selling for millions of $.
  - 1965: PDP-8
    * at $ 16,000, with 50,000 sold
    * important innovation: omnibus connecting the CPU, memory, I/O devices.
    * *http://en.wikipedia.org/wiki/PDP-8*

    The schematic representation of the PDP-8 omnibus is illustrated in Figure 5.

    .

- In this time, IBM followed 2 directions:
  - Expensive models, dedicated to scientific computing: 7090, 7094, see *http://en.wikipedia.org/wiki/IBM_7090*.
  - Cheaper models, dedicated to business computing: 1401, see *http://en.wikipedia.org/wiki/IBM_1401*

- Control Data Corporation:

  - 1964: CDC 6600
    * CPU able to perform parallel operations twice as fast as 7094,
    * small computers inside used to perform I/O operations "Snow White and the Seven Vertically Challenged People".
    * see *http://en.wikipedia.org/wiki/CDC_6600*
  - 6600 and its successors 7600, Cray-1 (designed by Seymour Cray) were dedicated to complex computations (supercomputers).

- Burroughs B5000 - built to run Algol 60, see *http://en.wikipedia.org/wiki/Burroughs_large_systems*.

Integrated Circuits (1965-1980)

- 1958 - the silicon integrated circuit (Robert Noyce,. co-founder of Intel): dozens of transistors can be put on a single chip, leading to smaller, faster computers.
- 1964 - IBM System/360 series:
  - both commercial and scientific applications,
  - replace the two separate strands of system design at IBM,
  - first commercial computers with microprogramming (emulation of both 7094 and 1401),
  - multiprogramming (many programs running in the same time on a processor),
  - huge address space (2^{24} bytes - 16 MB).
  - *http://en.wikipedia.org/wiki/IBM_360*
- DEC PDP-11 - popular at universities, *http://en.wikipedia.org/wiki/PDP_11*.

Very Large Scale Integration - VLSI (1980-?)

- In excess of tens of thousands of transistors on a single chip, making computers faster an cheaper ("the personal computer era").
- Originally, personal computer chips were available, consisting of:
  - printed circuit board,
  - CPUs (including Intel 8080),
  - cables, power supply,
  - 8 inch floppy disk,
  - no software (write your own, later CP/M on a floppy).
- Early PC's

  - IBM PC (1981 - public documentation for $ 49 leading to a PC clone industry),
  - Apple, Apple II (Steve Jobs, Steve Wozniak),
  - Amiga, Atari.
- Mid 1980's - RISC processors (replacing CISC processors).
- Mid 1990's - superscalar CPUs.

The Fifth Generation

- Japan's fifth generation computers (1982-1992),
  - *http://en.wikipedia.org/wiki/Fifth_generation_computer*
  - The aim was to create an "epoch making" computer, with supercomputer like performance and usable artificial intelligence capabilities,
  - The machine was to be built on top of massive databases, using a logic programming language (Prolog, extended) to access data in the range of 100 M - 1 G Logical Inferences per Second (LIPS).
  - The project failed, as fifth generation computers were made obsolete by the development of the CPU beyond the "obvious limitations" that motivated the project, the internet and the development of graphical user interfaces (GUI).
  - Although the project failed in its goals, it lead to a strong development of research networks, relations, etc.
- Ubiquitous computing/Low power/invisible computers (alternative fifth generation): computing integrated in everyday objects and activities.

Figure 6 gives a summary of important milestones in the development of the digital computer.

The Next Generation?

- 3D circuit design (cubes of transistors instead of chips).
- Optical computing (replace electronics by optics).
- Molecular computing (chemical-biological processes for computing):
  - experiments were made of 4bit computation in test tubes,
  - DNA computing,
  - new computing models - p-systems (IeAT Timisoara).
- Quantum computing, with special applications (quantum cryptography).

Year	Name	Made by	Comments
1834	Analytical Engine	Babbage	First attempt to build a digital computer
1936	Z1	Zuse	First working relay calculating machine
1943	COLOSSUS	British gov't	First electronic computer
1944	Mark I	Aiken	First American general-purpose computer
1946	ENIAC	Eckert/Mauchley	Modern computer history starts here
1949	EDSAC	Wilkes	First stored-program computer
1951	Whirlwind I	M.I.T.	First real-time computer
1952	IAS	Von Neumann	Most current machines use this design
1960	PDP-1	DEC	First minicomputer (50 sold)
1961	1401	IBM	Enormously popular small business machine
1962	7094	IBM	Dominated scientific computing in the early 1960s
1963	B5000	Burroughs	First machine designed for a high-level language
1964	360	IBM	First product line designed as a family
1964	6600	CDC	First scientific supercomputer
1965	PDP-8	DEC	First mass-market minicomputer (50,000 sold)
1970	PDP-11	DEC	Dominated minicomputers in the 1970s
1974	8080	Intel	First general-purpose 8-bit computer on a chip
1974	CRAY-1	Cray	First vector supercomputer
1978	VAX	DEC	First 32-bit superminicomputer
1981	IBM PC	IBM	Started the modern personal computer era
1981	Osborne-1	Osborne	First portable computer
1983	Lisa	Apple	First personal computer with a GUI
1985	386	Intel	First 32-bit ancestor of the Pentium line
1985	MIPS	MIPS	First commercial RISC machine
1985	XC2064	Xilinx	First field-programmable gate array (FPGA)
1987	SPARC	Sun	First SPARC-based RISC workstation
1989	GridPad	Grid Systems	First commercial tablet computer
1990	RS6000	IBM	First superscalar machine
1992	Alpha	DEC	First 64-bit personal computer
1992	Simon	IBM	First smartphone
1993	Newton	Apple	First palmtop computer (PDA)
2001	POWER4	IBM	First dual-core chip multiprocessor

Figure 6: Milestones in the development of the digital computer.

1.3 The Computer Zoo

Technological and Economic Forces Driving Computer Development

- The primary force is the ability of chip manufacturers to pack more and more transistors per chip every year:
  - Moore's Law (Gordon Moore, co-founder and chairman of Intel, 1965):

    "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years."("Cramming more components onto integrated circuits", Electronics Magazine 19 April 1965).
  - today's formulation: the number of transistors doubles every 18 months (and it was correct so far).
  - the estimation is that it will hold until around 2020 (when physical limits will be reached).
  - Moore's law drives a virtuous circle: better technology → smaller prices → more products → more companies → better technology.
  - Figure 7 illustrates the fact that Moore's law still holds today.

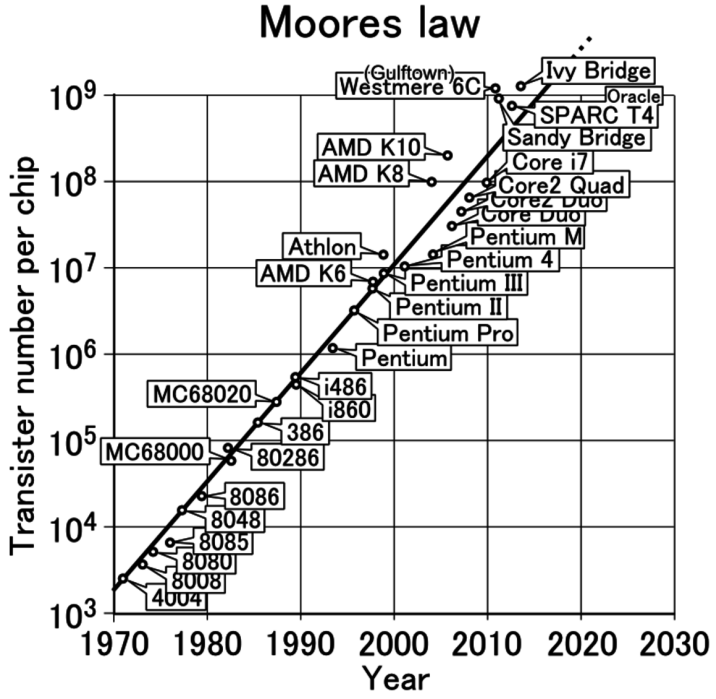


Figure 7: A representation of Moore's law (source: commons.wikimedia.org).

  - Nathan's first law of software (Nathan Myhrvold, former top executive, Microsoft):

    "software is as gas - it expands to fill the container holding it".

- Not all computer technologies develop as fast: storage capacity increases only about 50% per year.
- Communications, on the other hand saw spectacular development (high speed Internet).

The computer spectrum

- Several ways of using Moore's law:
  - Increase computer power at constant price.
  - Build the same computer for less and less.
  - Shrink the size of the hardware for constant power.
- Categories of computers:

Type	Cost ($)	Example application
Disposable computer	0.5	Greeting cards
Microcontroller	5	Watches, cars, appliances
Game console	50-200	Home/portable video games
Personal computer	500	Desktop or notebook
Server	5K	Network server
Collection of workstations	50-500K	Departmental minisupercomputer
Mainframe	5M	Batch data processing in a bank

Disposable computers

- To the low end, greeting cards, playing cheerful tunes.
- Possibly the most important development: RFID (Radio Frequency IDentification) chips:
  - smaller than 0.5 mm on edge, with 128 bit ID,
  - Applications:
    - labelling products in stores
    - animal ID (pets, farm animals, etc.),
    - vehicle tracking (controversial),
    - airline luggage,
    - cash marking (possible RFID in Euro notes),
    - *http://www.rfid.org*.
  - Critical design issues: Price, energy, application-specific performance.

Microcontrollers

- Microcontrollers = computers embedded in devices that are not sold as computers.
- They manage the devices and/or user interfaces.
- Found in a large variety of devices:
  - Appliances (clock radio, dishwasher, microwave, alarm).
  - Communication gear (cordless phones, cell phones, pagers).
  - Computer peripherals (printer, scanner, modem, CD ROM drive).
  - Entertainment devices (DVD, stereo, mp3 player).
  - Imaging devices (TV, digital cameras, photocopier).
  - Medical devices (X ray, MRI, digital thermometer).
  - Military weapon systems (missiles, etc.).
  - Shopping devices (vending machines, ATM).
  - Toys (talking dolls, radio-controlled cars, etc.)
- Critical design issues: Price, energy, application-specific performance.
- Example: a car could contain up to 50 microcontrollers (ABS, fuel injection, radio, GPS, etc), a plane more than 200.
- Compared to RFIDs, microcontrollers are complete computers (processor, memory, I/O capabilities).
- Types:
  - general purpose - complete computers,
  - special purpose (architecture geared to some application, e.g. multimedia).
- General purpose microcontrollers differ from ordinary computers:
  - extremely cost sensitive (in large number they may cost as little as $ 0.01 per unit),
  - operate in real time (get a stimulus and react to it),
  - physical constraints (designed with size restrictions in mind)

Game consoles

- Game consoles - normal computers with special graphics and sound capabilities, but limited software and little extensibility.
- Started out as low-end machines (pong on TV) but evolved into far more powerful systems (sometimes outperforming personal computers in certain dimensions).
- Examples (last generation): Microsoft XBOX 360, Sony Playstation 3, PSP, Nintendo Wii.
- Other characteristics:
  - Customized hardware: special processors, I/O devices.
  - Low cost - price supported by manufacturers (consoles sold at a loss, profits from games).
- See *http://en.wikipedia.org/wiki/Game_consoles*.
- Critical system design issues: Price-performance, energy, graphics performance.

Personal computers

- Personal computer is what people think in general when hearing "computer"
- Include desktop and notebook models.
- Some people call "PC" Intel CPU machines, "workstations" high-end RISC CPU machines (see also "Apple vs. PC") but they are essentially the same, conceptually.
- Other closely related machines: PDAs.
- Critical system design issues: Price-performance, energy, graphics performance.

Servers

- Servers are beefed up workstations, used for local networks or the Internet.
- Single processor and multiprocessor configurations.
- Architecturally not that different from personal computers, but typically much larger memories, storage, network bandwidth.
- Critical system design issues: Throughput, availability, scalability, energy.

Collections of workstations

- COW - Clusters of Workstations - multiple workstations collected together.
- Special software allows them to work on a single problem.
- The clusters are many time COTS (Commodity Off The Shelf) machines linked by high-speed networking hardware.
- These scale easily.
- One typical application: Internet Web server - server farms (hundreds, thousands of servers).
- Critical system design issues: Price-performance, throughput, energy proportionality.

Mainframes

- In many cases, mainframes are descendants of the IBM 360.
- They are not much faster than a powerful sever, but may have higher I/O capacity, higher storage.
- Very expensive, but many companies find it cheaper to pay once in a while for a mainframe, than update software.
- This has led to the Y2K problem (originating in the 1960-70s when COBOL programmers used only 2 digits for years).
- Andrew Tanenbaum predicts "the end of civilization as we know it at midnight on Dec. 31, 9999, when 8000 years worth of COBOL programs crash simultaneously."
- Critical system design issues: Price-performance, application-specific performance, throughput, energy proportionality.

1.4 Computer Families

Intel x86 -*

- 1968 - Robert Noyce, Gordon Moore, Arthur Rock form Intel Corporation (and sell $ 3000 worth of chips).
- *http://en.wikipedia.org/wiki/Intel_x86*
- Some notable milestones in the Intel (and related) processor development:

Chip	Date	MHz	Trans.	Memory	Notes
4004	4/1971	0.108	2300	640	First microprocessor on a chip
8008	4/1972	0.108	3500	16 KB	First 8-bit microprocessor
8080	4/1974	2	6000	64 KB	First general-purpose CPU on a chip
8086	6/1978	5–10	29,000	1 MB	First 16-bit CPU on a chip
8088	6/1979	5–8	29,000	1 MB	Used in IBM PC
80286	2/1982	8–12	134,000	16 MB	Memory protection present
80386	10/1985	16–33	275,000	4 GB	First 32-bit CPU
80486	4/1989	25–100	1.2M	4 GB	Built-in 8-KB cache memory
Pentium	3/1993	60–233	3.1M	4 GB	Two pipelines; later models had MMX
Pentium Pro	3/1995	150–200	5.5M	4 GB	Two levels of cache built in
Pentium II	5/1997	233–450	7.5M	4 GB	Pentium Pro plus MMX instructions
Pentium III	2/1999	650–1400	9.5M	4 GB	SSE Instructions for 3D graphics
Pentium 4	11/2000	1300–3800	42M	4 GB	Hyperthreading; more SSE instructions
Core Duo	1/2006	1600–3200	152M	2 GB	Dual cores on a single die
Core	7/2006	1200–3200	410M	64 GB	64-bit quad core architecture
Core i7	1/2011	1100–3300	1160M	24 GB	Integrated graphics processor

Figure 8: Key members of the Intel family.

Chip	**Date**	**Memory**	**Notable features**
4004	1971	640	First microprocessor on a chip.
8008	1971	16 KB	First 8 bit microprocessor.
8080	1974	64 KB	First general-purpose CPU on a chip.
8086	1978	1 MB	First 16 bit CPU on a chip.
8088	1979	1 MB	Used in IBM PC.
80286	1982	16 MB	Memory protection.
80386, AMD am386	1985	4 GB	First 32 bit CPU.
80486	1989	4 GB	Built-in 8 KB cache memory, RISC like pipelining, int
Pentium (Pentium MMX)	1993	4 GB	Superscalar, MMX.
Cyrix 6x86	1996	4GB	Register renaming, speculative execution.
Pentium Pro, AMD K5	1995	4GB	Microoperation translation, 2 level cache (only Pentiu
Athlon	1999	4GB	Superscalar FPU design.
AMD K6-2/3, Pentium II	1997	4 BG	Extra MMX instructions.
Pentium 4	2000	4 GB	Deeply pipelined, high frequency, SSE2, hyperthreadi
Pentium M	2003	4 GB	Optimized for low power.
Athlon 64/Opteron	2003	up to 1 TB	64 bit instruction set, on-die memory controller, hype
Core 2	2006	up to 1 TB	low power, multi-core, lower clock frequencies, SSE4.
AMD Phenom	2007	up to 1 TB	Monolithic quad-core, 128 bit FPU, SSE4a, modular c
Intel Atom	2008	4 GB/up to 1TB	Low power (netbook, nettop).
Core i7	2008	up to 1TB	Quad core.
AMD Bobcat	2011	as above	on-die GPU, low power.
Intel Sandy Bridge/Ivy bridge	2011-2012	as above	highly modular, on-die GPU.
Haswell/Skylake	2013-2016	as above (but more and better)	as above (but more and better)

Figure 8 gives another summary of the most important members of the Intel family.

UltraSPARC

- 1970's Unix could only run on minicomputers such as PDP 11;
- 1981 Andy Bechtolsheim SUN 1 (Stanford University Network) a personal computer that would run Unix (Sun Microsystems 1982);
  - these workstations (Sun 1, 2, 3) used Motorola CPUs;
  - the Sun workstations were more powerful than the PCs of the day and came equipped with an Ethernet connection and TCP/IP software;
- 1987 Sun decides to design its own CPU, based on the revolutionary new design from the University of California at Berkeley (RISC II):

Enters SPARC (Scalable Processor ARChitecture) the basis of Sun 4;

- Various manufacturers produced their own implementation of the open architecture:
  - MicroSPARC,
  - HyperSPARC,
  - SuperSPARC,
  - TurboSPARC,

  each of the above being compatible with the architecture (running the same programs);
- SPARC International Consortium manages the development of the SPARC architecture;
- The initial SPARC CPU: 32 bit machine 36 MHz, 1986 Sun/Fujitsu;
- 1995 UltraSPARC I: 64 bit architecture, 64 bit address space, 64 bit registers, but still able to run SPARC 32 bit instructions:
  - implementing the SPARC V9 specifications,
  - instructions designed to handle images, audio – VIS (Visual Instruction Set);
- 1997 UltraSPARC II;
- 2001 UltraSPARC III;
- 2003 UltraSPARC IV;
- 2006 UltraSPARC T1;
- 2007 (late) UltraSPARC T2;
- 2010 (late) SPARC T3.
- 2011 SPARC T4 (Oracle).
- 2013 SPARC T5 (Oracle).
- 2015 SPARC64 XIfx (Fujitsu), SPARCM7 (Oracle).

ARM

- Advanced RISC design, initially Acorn Computer RISC design, later Advanced RISC Machine.
- Timeline:
  - 1985: ARM2 (popular in schools UK, Ireland, Australia, New Zealand),
  - 1993: ARM 610 (Apple Newton),
  - mid 90's: strongArm (ultrafast - 233MHz, ultra low power - 1W),
  - 1994: ARM7 (still used today)
  - 2011: 64 bit ARM architecture.
- ARM does not manufacture processors, but licenses them.
- ARM is the most widely used processor (over 50 billion processors by 2015, growing).
- Typical applications: hand-held devices, home appliances, etc.
- Some characteristics: RISC design, low power, hardware support for Java bytecode.

Java

- Sun Microsystems: mid 1990's designs Java (inspired by C++), aiming at crossplatform compatibility;
- JVM (Java Virtual Machine): memory of 32 bit words, 226 instructions;
- Java programs are compiled for JVM;
- To run compiled Java code on a machine, one needs an interpreter (usually written in C i.e. available virtually everywhere, but generally slow);
- JIT (Just in Time) compilers:
  - JVM to machine compilers, usually found in web browsers,
  - using a JIT solution may cause delays in execution;
- picoJava II (1998)
  - a chip architecture that implements JVM,
  - designed for embedded devices (i.e. low cost, way under 50$),
  - various implementations available.

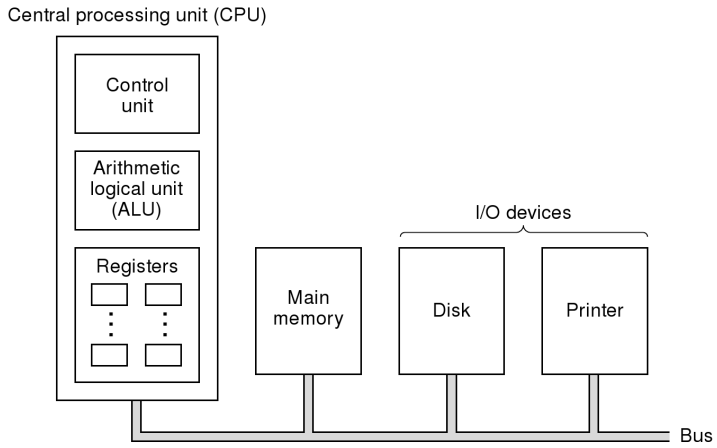


Figure 9: The organization of a simple computer.

2 Computer Systems Organization

Computer organization

- A digital computer is an interconnected system of processors, memories and input/output devices.
- In the following we focus on each of these 3 categories.

Figure 9 illustrates the organization of a digital computer.

- CPU (Central Processing Unit) - the "brain" of the computer.

  Its function is to execute the programs stored in the main memory by

  - fetching instructions,
  - examining them,
  - executing them one after the other.

- A bus:

  - connects the components of a computer,
  - is a collection of wires (parallel, serial) for transmiting address, data and control signals,
  - external buses connect the CPU to the memory and the I/O devices,
  - internal buses are found inside the CPU.

A modern computer today

	Desktop workstation	Mobile
Processor	Core i7X 990x 3.20 GHz Quad	A5 (ARM Cortex-A9 MPCore 1GHz)
Memory	36 GB DDR3	512 MB DDR2
Secondary storage	2 x 2 TB SATA 7200 rpm HDD	64 GB SSD
Keyboard, mouse	Optimus Maximus	none
Videocard	AMD Radeon HD6870 2 GB	Integrated on-chip (PowerVR SGX543MP2)
Monitor	2 x 30" LCD	9.7" LCD multitouch
Network card	Intel(R) PRO/1000 EB	Wi-Fi, 3G
Optical drive	Blu-ray Disk Burner	-

2.1 Processors

CPU Organization

- Control Unit
  - fetches instructions from the main memory and determines their type.
- ALU
  - performs operations needed to carry out the instructions,
  - e.g.: addition, boolean AND.
- Registers
  - small high-speed memory used to store temporary results and certain control information,
  - general purpose and special purpose, e.g.:

    Program Counter (PC) - points to the next instruction to be executed,

    Instruction Register (IR) - holds the instruction that is currently executed.

Data Path

- The registers, together with the ALU and several buses form the data path.
- Instructions:
  - register-memory
    - fetch words from memory,
    - store words back into memory.
  - register-register (data path cycle)
    - fetch operands from registers into the input registers,
    - bring the content into the ALU,

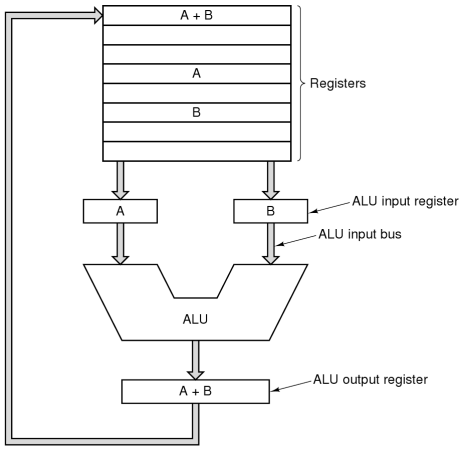


Figure 10: The data path of a typical von Neumann machine.

  - perform operation into the ALU,
  - store result back intro a register.

  Example: Figure 10 illustrates an addition operation on a typical von Neumann machine.
- The data path cycle is the core of the CPU.

Instructions execution

- The CPU executes the instructions in a series of small steps:
  1. Fetch the next instruction from memory into an instruction register.
  2. Change the program counter to point to the next instruction.
  3. Determine the type of instruction just fetched.
  4. If the instruction uses a word in the memory, determine where it is.
  5. Fetch the word, if needed, into a CPU register.
  6. Execute the instruction.
  7. Go to step 1 to begin executing the next instruction.
- The above sequence is the fetch-decode-execute cycle, central to the operation of all computers.
- The way the CPU works can be described into some programming language. Figure 11 illustrate such an interpreter.

```
public class Interp {
   static int PC;                  // program counter holds address of next instr
   static int AC;                  // the accumulator, a register for doing arithmetic
   static int instr;               // a holding register for the current instruction
   static int instr_type;          // the instruction type (opcode)
   static int data_loc;            // the address of the data, or −1 if none
   static int data;                // holds the current operand
   static boolean run_bit = true;          // a bit that can be turned off to halt the ma

   public static void interpret(int memory[ ], int starting_address) {
      // This procedure interprets programs for a simple machine with instructions
      // one memory operand. The machine has a register AC (accumulator), used
      // arithmetic. The ADD instruction adds an integer in memory to the AC, for e
      // The interpreter keeps running until the run bit is turned off by the HALT ins
      // The state of a process running on this machine consists of the memory, th
      // program counter, the run bit, and the AC. The input parameters consist of
      // of the memory image and the starting address.

      PC = starting_address;
      while (run_bit) {
         instr = memory[PC];   // fetch next instruction into instr
         PC = PC + 1;          // increment program counter
         instr_type = get_instr_type(instr); // determine instruction type
         data_loc = find_data(instr, instr_type);// locate data (−1 if none)
         if (data_loc >= 0)    // if data_loc is −1, there is no operand
            data = memory[data_loc];     // fetch the data
         execute(instr_type, data);      //execute instruction
      }

   }

   private static int get_instr_type(int addr) { ... }
   private static int find_data(int instr, int type) { ... }
   private static void execute(int type, int data){ ... }
}
```

Figure 11: An interpreter for a simple computer (written in Java).

Interpretation of instructions

- Instruction interpreter - a program that can imitate the functions of a CPU (fetch, examine, execute).
- Hardware-software equivalence!
- Design decision for a new language L:
  - direct hardware implementation,
  - implementation of a software interpreter,
  - hybrid solutions.
- Instructions were initially simple (1950's), then more and more complex.
- Transition to complex instructions: sequence of simple instructions occuring frequently or special cases (floating point, array operations).
- Hardware implementation of complex instructions → more powerful computers.
- Economic factors:
  - hardware instructions - higher cost (Cray supercomputers).
  - solution: interpretation.

RISC vs. CISC

CISC (Complex Instruction Set Computer) (term coined later, in opposition to RISC.)

- since 1950's the instructions were interpreted,
- more and more instructions (DEC VAX: several hundred instructions, more than 200 ways to specify operands),
- interpretation provided a way to add new instructions quickly,
- interpretation provised easy bux fixing,
- interpretation ensured backward compatibility.

RISC (Reduced Instruction Set Computer)

- 1980, David Patterson and Carlo Séquin (Berkeley): VLSI CPU without interpretation,
- coined the term RISC and named the CPU RISC I (→ SPARC),
- 1981 John Hennessy (Stanford): MIPS CPU (→ MIPS),
- no backward compatibility to clog the design,

- key idea: instructions can be issued quickly and executed in hardware (too many in comparison with CISC instructions in order to be outperformed).

RISC vs. CISC ?

- too much invested in CISC software, hardware, RISC has not taken over,
- hybrid approaches (CISC architecture with RISC core - Intel Pentium + successors):

  simple (and more common) instructions executed in one data path cycle,

  complex instructions (occurring rarely) interpreted in the usual way.

Design principles for modern computers

- **All instructions directly executed by hardware.**
  - CISC instructions should be broken down into RISC instructions.
- **Maximize the rate at which the instructions are issued**.
  - it is less important how long instructions will take to execute,
  - parallelism could play an important role.
- **Instructions should be easy to decode.**
  - instructions should be regular, with fixed length and a small number of fields.
- **Only `Loads` and `Stores` should reference the memory**.
  - operands for all other operations should come from and return to registers.
- **Provide plenty of registers**
  - access to memory is slow → the more registers, the better.
- Attention: The above principles are relative to the current technological resources and limitations!

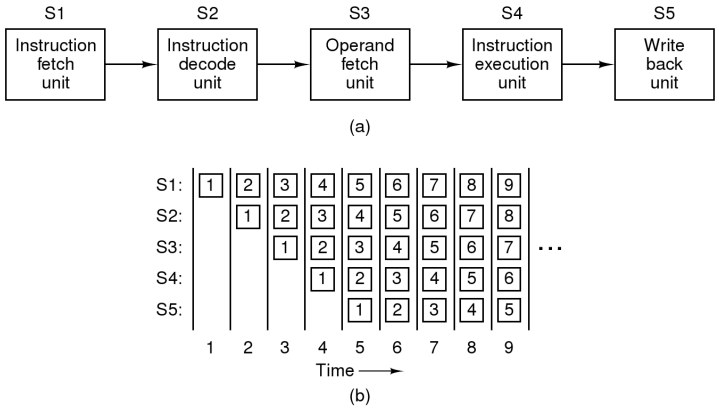


Figure 12: (a) A five stage pipeline. (b) The state of each stage as a function of time (nine clock cycles).

Improving computer performance

- Brute force: make chips faster by increasing the clock speed. (Limited by the technological development at a fixed moment in time.)
- Parallelism: doing more things at once - get more performance for a given clock speed.
  - Instruction level parallelism - get more instructins per second out of the machine.
  - Processor (thread) level parallelism - use more processors to share the work on the same problem.
  - Data level parallelism - spread data over multiple system (e.g. "cloud").

Instruction-level parallelism: pipelines

- Pipelining: divide instruction execution into small steps, each supported by a dedicated piece of hardware (as illustrated in Figure 12).

Instruction level parallelism: superscalar architectures

- "One pipeline good, two pipelines better."
  - pairs of instructions that do not conflict over resources or do not depend on eachother can be sent on different pipelines,
  - correctness of this process is ensured by the compiler, or using extra hardware,

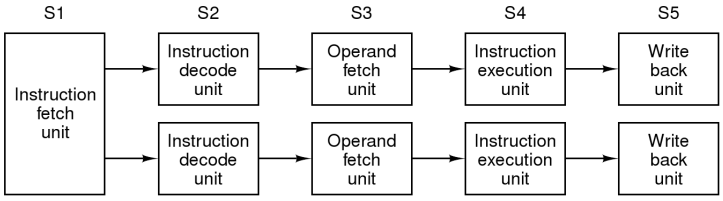


Figure 13: Dual five-stage pipeline with a common instruction fetch unit.

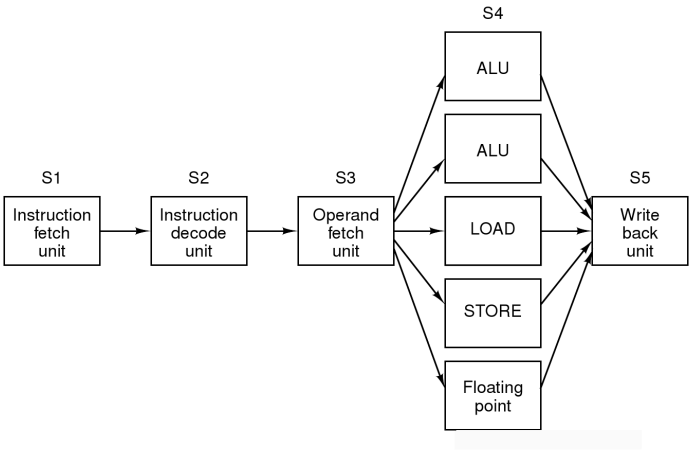


Figure 14: A superscalar processor with five functional units.

  - Example: Intel processors (Pentium):
    * the u pipeline: arbitrary instructions,
    * the v pipeline: integer and floating-point instructions.
    * this led to important performance gains (Pentium up to 2 times faster than 486).
  - Figure 13 illustrates the use of two pipelines.

- "Two pipelines good, four pipelines better"?

  Actually, no - too much hardware duplication.

- For high-end CPUs: one pipeline, but with multiple functional units, as illustrated in Figure 14.

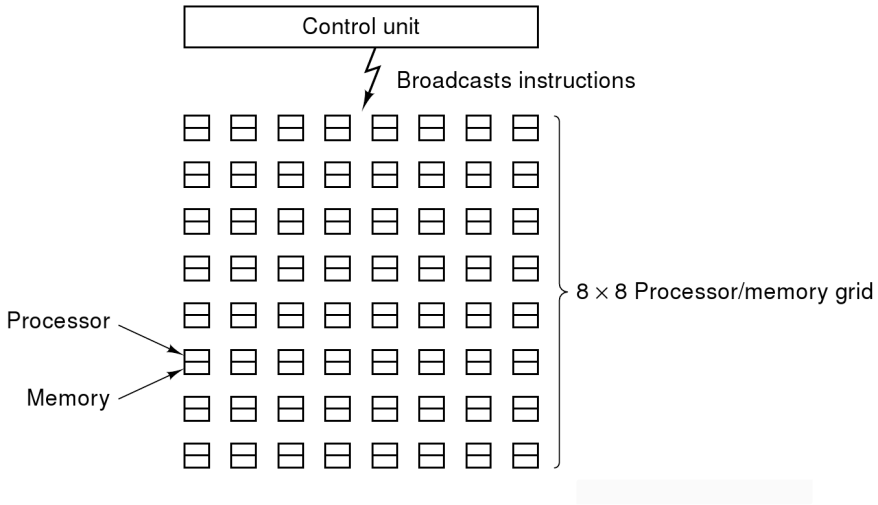


Figure 15: An array processor of the ILLIAC IV type.

Processor-level parallelism: array computers

- Array processors - large number of identical processors, performing the same sequence of instructions on different sets of data.
- ILLIAC IV, 1972 (4 quadrants of 8 x 8 square grid of processor-memory elements), only 1 quadreant built (cost overrun 4 times) - 50 megaflops (mil. floating point ops / second).
- Architecture known as SIMD (Single Instruction-stream Multiple Data), different form the standard von Neumann architecture.
- The design of an ILLIAC IV quadrant is illustrated in Figure 15.
- A modern incarnation is the Fermi GPU, illustrated in Figure 16.
- Vector processor - similar to the array processors, with the difference of having a heavily pipelined adder, special registers - vector registers (Cray-1, 1974).
- No array computers are in use today, but the same principle is used in Intel x86-* processors for multimedia instructions (MMX, SSE).

Processor-level parallelism: multiprocessors/multicomputers

- Multiprocessors - systems with multiple full-blown CPUs.

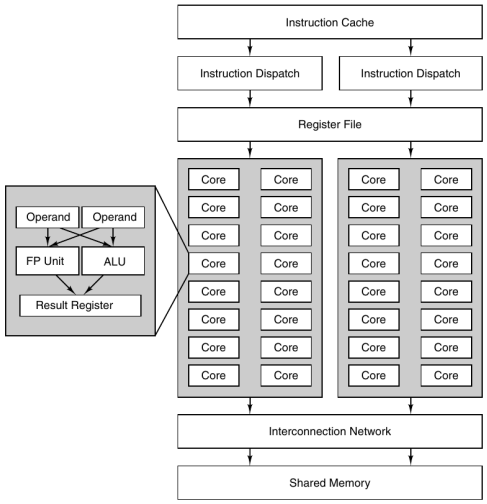


Figure 16: SIMD core of the Fermi GPU.

  - easy to build for small number of processors (up to 64), difficult for more.
  - Figure 17 illustrates possible implementation schemes for multiprocessor systems.

- Multicomputers - systems of interconnected computers, each with its memory.
  - several topologies for the connections: 2D, 3D, grid, trees, rings,
  - up to 10000 CPUs.
  - Hybrid approaches - multicomputers that simulate a shared memory.

Processor clock

- Instruction execution is driven by a high-frequency processor clock.

  e.g.
  - 3.5 GHz computer (3.5 billion Herz),
  - the clock ticks 3.5 billion times a second,

- Execution time (clock cycles):
  - instructions may take 1, 2, 3, ... clock cycles,
  - different instructions have different power,

- processor speed is not a measure of processor power (**megaherz/gigaherz myth**).

  Examples:

  * AMD vs. Intel,
  * Intel vs. PowerPC,
  * Intel Pentium M vs. Pentium 4.

2.2 Primary Memory / Secondary Memory / Input/Output (Old Slides)

The Primary Memory

• Memory = the part of a computer where programs and data are stored.
• Primary memory (fast, small, volatile) vs. secondary memory (slow, large, persistent).
• Synonyms (British) – storage, store (more and more used to refer to disk storage).

Bits
• basic unit of memory (holding 0 or 1);
• binary arithmetic "more efficient" (= easier to distinguish between 2 physical values rather than between 10);
• Example: BCD(*) (Binary Coded Decimal) vs. binary:

1944:
- BCD: 00001 1001 0100 0100
- binary: 0000011110011000

BCD: 0-9999
binary: 0-65,535

(*) [use 4 bits to represent one digit, 16 combinations possible, 6 not used, the rest 0 ,..., 9]

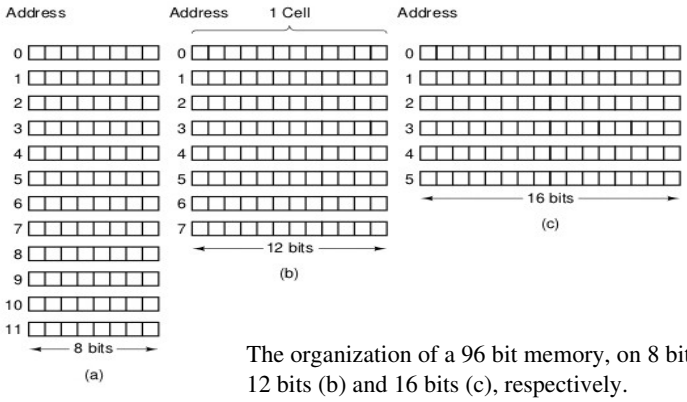
• Scenario: invention of a highly reliable electronic device to store 0, ..., 9.
- 4 digits: 0-9999;
- in binary, 4 digits: 0-15;

[in this case, the binary representation is less efficient]

Primary Memory

Memory Addresses

- memories consist of a number of cells (locations) – used to store information;
- each cell has a number, which gives its address;
- if an address is expressed on m bits, the maximum number of addressable cells is 2^m.



The organization of a 96 bit memory, on 8 bits (a), 12 bits (b) and 16 bits (c), respectively.

Primary Memory

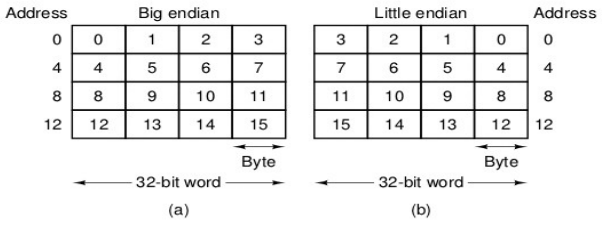
Memory Addresses (continued)

- a cell = smallest addressable unit;
- most computer manufacturers have standardized a 8-bit cell, called byte;
- bytes are grouped into words;
- word = instructions operate on entire words

  32 bit machines will have 32 bit registers;
  64 bit machines will have 64 bit registers.

Ordering bytes:
- big endian, left-to-right (SPARC, IBM mainframes);
- little endian (b), right-to-left (Intel).

- Problems with data transfer from one format to the other!



Units for Measuring Memory

Unit	Symbol	Bytes
byte		$2^0 = 1$ byte
kilobyte	KB	$2^{10} = 1024$ bytes
megabyte	MB	$2^{20} = 1024$ KB
gigabyte	GB	$2^{30} = 1024$ MB
terabyte	TB	$2^{40} = 1024$ GB

The multiplication factor for the next unit is $1024 = 2^{10}$.

Error Correcting Codes

- memories can occasionally produce errors;
- to avoid such problems, memories use error-detecting or error-correcting codes;
- an n bit word (codeword) usually contains m data bits and r redundant bits:

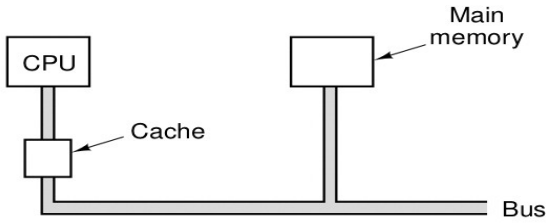
$$n = m + r$$

- Hamming distance = the minimum number of bits that are different between 2 codewords;
- Example:

  10001001 and 10110001 have the Hamming distance 3;
- with Hamming distance d, d single bit errors are required to convert from one into another;
- for a n bit codeword ($n = m + r$), there are 2^n codewords that can be formed, of which only 2^m are legal; whenever the computer detects an illegal codeword, an error has occurred;
- Example: parity bits.

Cache Memory

- historically, CPUs have always been much faster than memories;
- this leads to inefficiencies in execution (CPU has to wait for the data from the memory);
- technologically, it is possible to have fast enough memories, but this is not economically viable;
- solution: use a small amount of very fast – cache memory at the CPU level;
- the most heavily used words are kept in the cache;
- CPU first looks in the cache, and only then in the main memory;
- logically, the cache lies between the main memory and the CPU, in practice it can be located in several other places.



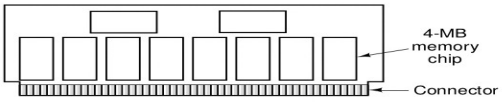
Cache Memory (continued)

- programs run faster if the data they need is in the cache and not in the main memory;
  HOW TO ACHIEVE THAT ?
- locality principle: memory references made in a short time interval tend to use only a small fraction of the total memory;
- if a word is read or written k times in a short interval, then 1 slow reference to the main memory is needed, and k-1 references to cache – the bigger the k the better;
- memories and cache are organized in cache lines: whenever a word is needed from the memory, the whole cache line will be loaded;
- cache design issues:
  - size of cache: 16KB, ..., 2 MB, 6MB – the bigger the more expensive the CPU;
  - organization of cache;
  - whether instructions and data are kept in different caches:
    - unified cache: simpler to build, balance between instructions and data;
    - split cache (Harvard architecture): allows parallel accesses, good with pipelines;
  - number of caches: not uncommon to have one on the CPU, one off chip but still in the same package, and a third one further away.

Primary Memory

Memory Packaging

- initially, memory chips were separate units (1Kb-1Mb), plugged directly in sockets;
- at present, memory chips are packaged into circuit boards, which are sold as units;
- memory units:
  - SIMM (Single Inline Memory Module):
    - 72 connectors;
    - 32 bits transfers;
    - paired for 64 bit transfers, each doing half;
    - typical sizes: 32MB, 64MB, ...



  - DIMM (Double Inline Memory Module):
    - 84 gold-plated connectors;
    - 64 bits transfer;
    - typical sizes: 64MB, ...
    - variants: SO-DIMM (Small Outline DIMM) – laptops.
- Usually no error detecting/correcting features, as average error rate: 1/10 years.

Primary Memory

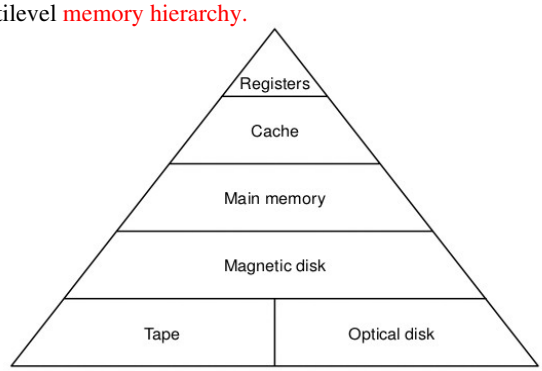
RAM Types

- SRAM (Static RAM)
  - used for cache, motherboard memories, digital signal processing;
  - retains its contents as long as power remains applied;
- NVRAM (Non-Volatile RAM)
  - user-programmable memory chip whose data is retained when power to the chip is turned off;
  - modems, MP3 players;
- DRAM (Dynamic RAM)
  - stores each bit of data in a separate capacitor -> leaks;
  - use refresh logic to refresh the memory;
  - bigger, better, faster, more:
    - Fast Page Mode DRAM;
    - EDO RAM (Extended Data Out DRAM);
    - SDRAM (Synchronous DRAM) ;
    - DDR SDRAM (Double Data Rate Synchronous DRAM);
    - RDRAM (Rambus DRAM).

The Secondary Memory

Memory Hierarchies

- The trouble with the registers, cache, main memory is that although more and more is available, there isn't enough to go around to store all that needs to be stored.
- Example: 50.000.000 books (U.S. Library of Congress) each with 1 MB of textand 1 Mb pictures would fit on about 100 TB.
- Solution: use a multilevel memory hierarchy.



Secondary Memory

Memory Hierarchies (continued)

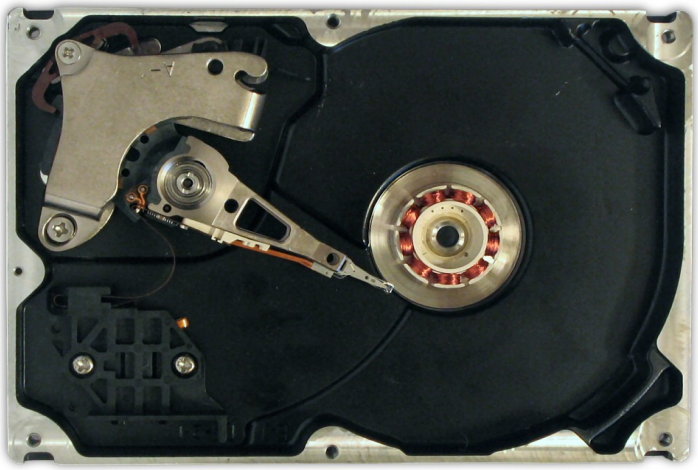
- On the 5 level memory hierarchy, as we move down, 3 parameters increase:
  1. the access time:
     - registers (nanoseconds),
     - cache (small multiple of registers),
     - main memory (tens of nanoseconds),
       [~~~~~ huge gap ~~~~~]
     - disc access (at least 10 milliseconds),
     - tape, optical disc (seconds, taking into account handling).
  2. the storage capacity:
     - registers (~ 128 bytes),
     - cache (a few MB),
     - main memory (~ 1 GB),
     - magnetic disks (hundreds of GB, now TB),
     - etc...
  3. number of bits/$ (or favorite currency)
     - prices change rapidly,
     - main memory: $/ MB;
     - magnetic disks: cents/ MB;
     - magnetic tape: $/GB.

Secondary Memory

Performance Comparison

Operation	Clock cycles
CPU instructions	1-3
Register access	1
Cache access	1
Main memory access	10
Disk seek time	10,000,000

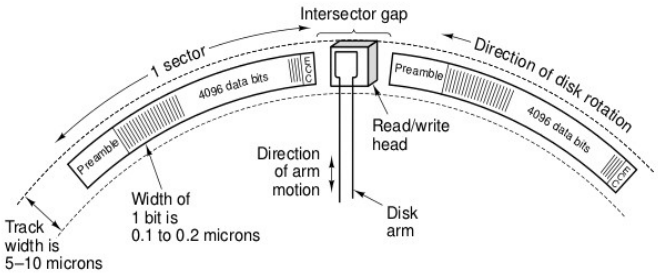
The Hard Disk



Magnetic Disks

- magnetic disk =
  - one or more aluminum platters with a magnetizable coating;
  - a disk head floating above the surface on an air cushion, at the end of an arm;
    - when a positive or negative current passes through the head, it magnetizes the surface of the platter (write);
    - when the head passes over the platter area, a positive or negative current is induced on the head (read);

The geometry of a disk track (circular sequence of bits written as the disk makes a complete rotation):

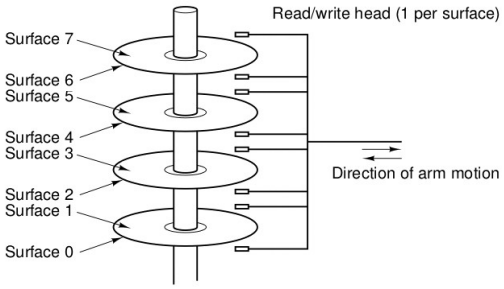


Organization of Magnetic Disks

- tracks are divided up in sectors of fixed length;
- sectors = preamble (allows synchronization of the head) + data (512 bytes) + error correcting code (ECC);
- EEC: Hamming code (single errors) or Reed-Solomon code (multiple errors);
- between each sector there is an intersection gap;
- unformatted disk capacity – total of sectors, incl. preambles, data, EEC, intersection gaps;
- formatted disk capacity – just the data capacity;
- storage capacity is influenced by:
  - radial density: number of tracks per radial centimeter (800-2000);
  - linear bit densities: number of bits per track centimeter (~ 100.000);
- most disks consist of several platters stacked vertically, each with its own arm and head;
- all arms are gaged together, so they move to different radial positions all at once;
- cylinder = the set of tracks at a given radial position;

Performance

- a disk with 4 platters:



- factors that influence the performance of a magnetic disk:
  - seek: the time needed to move the heads at a certain radial position (5-15 ms);
  - rotational latency: the time needed for the desired sector to be placed under the head (4-8 ms);
- rotation speeds: 4200, 5400, 7200 ... RPM;
- transfer rates 5-20 MB/sec;
- burst rate: transfer rate when the head is over the first bit in the sector;
- sustained rate: much smaller than the burst rate.

Secondary Memory

Disk Controllers

- disk controllers are chips (sometimes full CPUs) associated with each drive;
- their task:
  - accept commands from software: READ, WRITE, FORMAT;
  - control the arm motion;
  - detect, correct errors;
  - buffering of multiple sectors;
  - caching sectors;
  - remapping bad sectors.

Floppy Disks

- miniature disks (difference: head on the surface, small capacity – up to 1.44 MB);
- cheap, unreliable !!; [always make multiple copies]
- surprisingly not yet extinct (spring-summer 2008, in fact essential to the economy!!!!)
- (retro-cool? - not likely).

Secondary Memory

IDE disks

- Integrated Drive Electronics – mid 1980s
  - controller integrated with the driver;
  - backward compatible (BIOS calling conventions);
  - because of backward compatibility, it could address with 4 bits for the head, 6 bits for the sectors, 10 bits for the cylinder max disk: 16 heads, 63 sectors, 1024 cylinders ~ 1,032,192 sectors ~ 528 MB;
- EIDE (Extended IDE)
  - support for a second addressing scheme, LBA (Logical Block Addressing);
  - 2^{24}-1 sectors;
  - the controller has to convert LBA addresses to head, sector, cylinder addresses;
  - other improvements: ability to control up to 4 drives, higher transfer rate;
  - can also control CD-ROM drives, tape drives, etc.
- ANSI standard: ATAPI (Advanced Technology Attachment Packet Interface);
- This technology is comparatively cheap, widely used for the garden variety PCs;
- Drawback: internal only (limitations on cable length);
- The terms (E)IDE and ATA are used interchangeably.

Secondary Memory

S-ATA disks

- Successor of ATA (or IDE, now retrospectively called P-ATA);
- SATA/150 (2003)
  - 1.5 GHz (~1200 MBits/s transfer rate due to encoding 8B/10B),
  - new cable design (less clutter inside boxes),
  - hot-swappable,
  - native command queuing (NCQ) – handle several I/O requests;
- SATA/300 (2004)
  - 3.0 GHz;
- e-SATA:
  - SATA technologies can be used for external devices,
  - more performance than current external solutions,
  - yet to become widely available.

Secondary Memory

SCSI Disks

- Small Computer System Interface ("scuzzy")– 1986
- since then, increasingly fast versions have been standardized:
  - SCSI 1: 8 data bits, 5 MHz bus, 5 MB/sec transfer,
  - Fast SCSI: 8 data bits, 10 MHz bus, 10 MB/sec transfer,
  - Wide Fast SCSI: 16 data bits, 10 MHz bus, 20 MB/sec transfer,
  - Ultra SCSI: 8 data bits, 20 MHz bus, 20 MB/sec transfer,
  - Wide Ultra SCSI: 16 data bits, 20 MHz bus, 40 MB/sec transfer,
  - Ultra2 SCSI: 8 data bits, 40 MHz bus, 40 MB/sec transfer,
  - Wide Ultra2 SCSI: 16 data bits, 40 MHz bus, 80 MB/sec transfer,
  - Ultra3 SCSI: 16 data bits, 40 Mhz DDR bus, 160 MB/sec transfer, CRC, error correction (1999),
  - Ultra-320 SCSI: 16 data bits, 80 Mhz DDR, 320 MB/sec,
  - Ultra-640 SCSI – pushing the hardware limits, very short cables, impractical,
  - New research and development: Serial SCSI (various approaches);
- standard disk controllers for UNIX workstations (Sun, HP, SGI), Macs, high-end Intels;
- SCSI: controller + several devices (hard-disks, CD-ROMs, scanners, etc.)
  - each device has a unique ID;
  - each device has 2 connectors: one for input and one for output;
  - each output connected to the input of the next device, last device terminated;
- SCSI controllers and peripherals act as initiators or targets:
  - commands and responses occur in phases,
  - arbitration allows all the devices to run at once (as opposed to EIDE, where only one device can run at a time;

Secondary Memory

SUMMARY: Disk Performance

SCSI 1	12.0 Mbit/s	**1.5 MB/s**
Fast SCSI 2	80 Mbit/s	**10 MB/s**
Fast Wide SCSI 2	160 Mbit/s	**20 MB/s**
Ultra DMA ATA 33	264 Mbit/s	**33 MB/s**
Ultra Wide SCSI 40	320 Mbit/s	**40 MB/s**
Ultra DMA ATA 66	528 Mbit/s	**66 MB/s**
Ultra-2 SCSI 80	640 Mbit/s	**80 MB/s**
Ultra DMA ATA 100	800 Mbit/s	**100 MB/s**
Ultra DMA ATA 133	1064 Mbit/s	**133 MB/s**
Serial ATA (SATA-150)	1200 Mbit/s	**150 MB/s**
Ultra-3 SCSI 160	1280 Mbit/s	**160 MB/s**
Fibre Channel	**800 or 1600 Mbit/s**	100 or 200 MB/s
Serial ATA (SATA-300)	2400 Mbit/s	**300 MB/s**
Ultra-320 SCSI	2560 Mbit/s	**320 MB/s**
Ultra-640 SCSI	5120 Mbit/s	**640 MB/s**

Secondary Memory

RAID

• Redundant Array of Inexpensive [/Independent] Disks:
- as opposed to SLED (Single Large Expensive Disk);
- specific disk organization can improve performance/reliability;
- a RAID controller + many disks (RAID SCSI obvious choice);
- visible to the system as a single disk.

• RAID levels:

Level 0:
- distribute data over multiple disks: stripping;
- no parallelism, reliability potentially worse than SLED;

Level 1:
- RAID 1 + backup;
- distributed (fast) read / write twice - bad;
- backup disks (fault tolerance);

Level 2:
- working on word basis;
- distribute bits over a large number of disks (and add Hamming code);

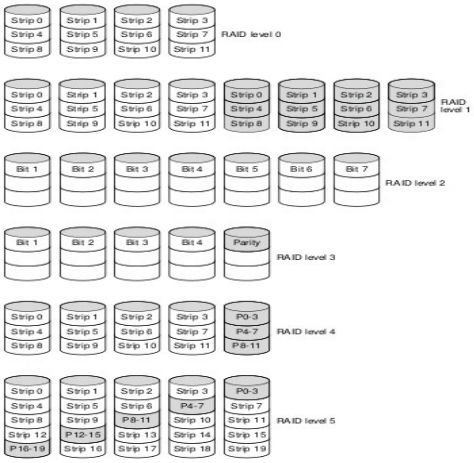
Level 3:
- simplified version of level 2: use parity bit instead;
- fault tolerance in case of 1 drive failing;

Level 4:
- like level 0, with 1 disk used for parity;
- fault tolerance, but poor performance for small updates;

Level 5:
- like level 0, with parity bits distributed among the disks;
- good performance, but in event of a crash, reconstructing the content is difficult;

Secondary Memory

RAID Levels



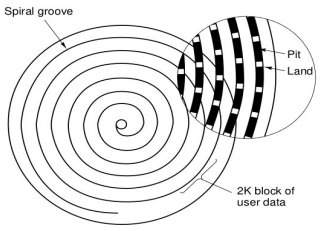
Other RAID configurations

- combinations are possible (e.g. RAID 1+0/0+1/5+0);
- proprietary RAID formats: RAID 1.5, 7, S, Z, ServerRAID 1E, etc.

Secondary Memory

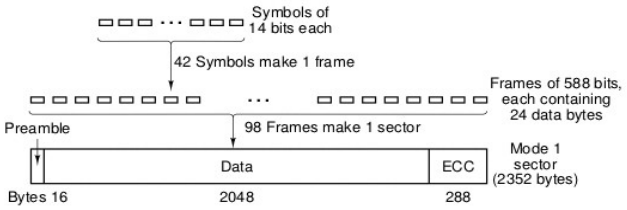
CD-ROMs

- 1980 Philips + Sony introduce the CD (Compact Disk);
  - the red book (ISO 10149):
    - size: 120 mm across, 1.2 mm thick, 15 mm hole;
    - high power infrared laser burns 0.8 micron holes in a coated glass master disk;
    - a mold is made from the master -> copies;
    - playback: low-power laser diode – infrared light – detects pits and lands;
    - pits and lands written on a continuous spiral (22,128 revolutions, 5.6 km long);
    - rotation rate 530RPM-200RPM;
    - the first commercially successful digital storage medium;



CD-ROMs (continued)

• 1984 Sony and Philips CD-ROM (Compact Disk – Read Only Memory)
- yellow book- data format specifications:
- encode every byte on 14 bites – ECC;
- logical data layout:

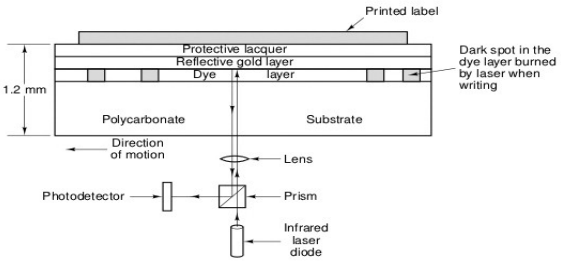


- mode 1: above;
- mode 2: for applications that do not need error correction;
- CD-ROM speeds: 1x – [153,600 bytes/s – 175,200 bytes/s];
- green book (1986 – Philips): graphics, audio, video;
- CD-ROM filesystem: ISO 9660 – any OS.

Secondary Memory

CD-Recordables

• orange book (1989):
- reflective gold layer instead of aluminum;
- a layer of dye: cyanine (green) or pthalocyanine (yellow-orange);
- CD-R laser –write mode: high power (8-16 mW) to melt the dye;
- CD-R laser – read mode: 0.5 mW;



- CD-Rs allow incremental writing (multisession): multiple VTOCs (Volume Table of Contents);
- primarily intended for data backup and recovery;

CD-Rewritables

- CD-RW
  - use same media as CD-R, but an alloy of silver, indium, antimony and tellurium instead of dye;
  - CD-RW lasers: high-power to melt the alloy (write), med-power to melt the alloy back to land (format, erase), low-power (read);

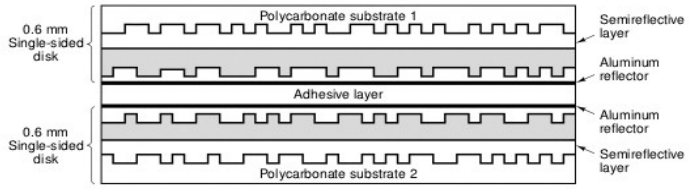
CD-* Conclusions, Summary

- Size: 650 – 700 – 800 MB;
- Copyright issues:
  - music, software industry: write protection:
    - multigigabyte file lengths;
    - introduce errors;
    - nonstandard gaps.

DVDs

- "digital video/versatile disk" Nov. 1996 (Japan), Mar. 1997 (US);
- A DVD can contain:
  - DVD-Video (movies (video and sound)) [regions];
  - DVD-Audio (high-definition sound);
  - DVD-Data (containing data);
- The disc medium can be:
  - DVD-ROM (read only, manufactured by a press)
  - DVD+R/RW (R=Recordable once, RW = ReWritable)
  - DVD-R/RW (R=Recordable once, RW = ReWritable)
  - DVD-RAM (random access rewritable)
- DVD formats:
  - DVD-5: single sided, single layer, 4.7 GB
  - DVD-9: single sided, double layer, 8.5 GB
  - DVD-10: double sided, single layer on both sides, 9.4 GB;
  - DVD-14: double sided, double layer on one side, single layer on other, 13.2 GB;
  - DVD-18: double sided, double layer on both sides, 17.1 GB;

DVDs (continued)



Example: a double-sided, dual layer DVD disk;

- DVDs achieve higher capacity than CDs by using:
  - smaller pits (0.4 microns vs. 0.8 microns for CDs);
  - tighter spiral (.74 microns vs. 1.6 microns for CDs);
  - red laser (0.65 microns vs. 0.78 microns for CDs).

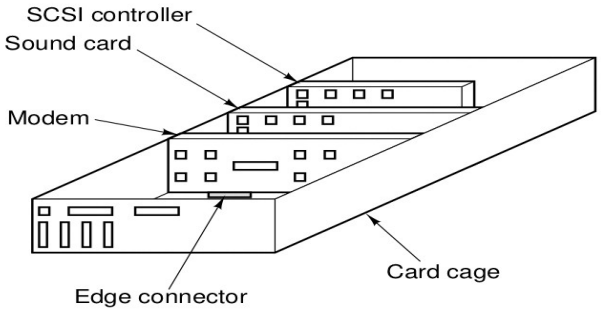
DVDs, new formats, based on "blue" lasers

- Blu-Ray (Sony, Disney, Apple, and more):
  - 23.3/25/27 GB (single layer), 46.6/50/54 GB (dual layer) [12 cm]
  - 7.8 GB (single layer), 15.6 GB (dual layer) [8 cm]
- HD DVD (Toshiba, Intel, Microsoft, and more) [as of spring 2008, dropped] :
  - 15 GB (single layer), 30/45 GB (dual layer) [12 cm].

INPUT / OUTPUT
Buses and Interfaces

Input / Output

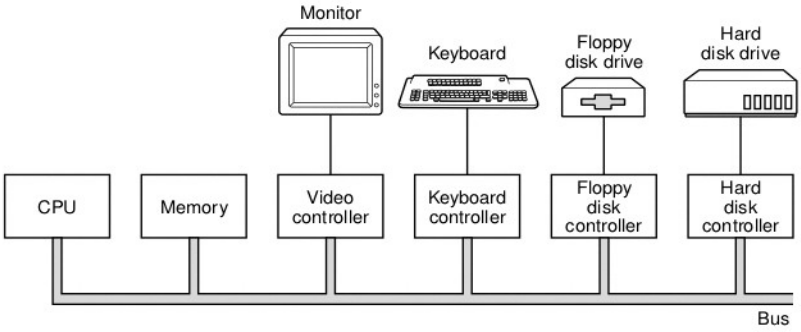
The Physical Structure of a Personal Computer



- a metal box with a large printed circuit board, the motherboard which contains...
- the CPU chips,
- slots in which DIMM modules can be clicked,
- chipsets,
- sockets for I/O boards,
- bus(es) – high speed (for modern I/O boards), low speed (for older I/O boards),
- code in ROM.

Input / Output

The Logical Structure of a Personal Computer



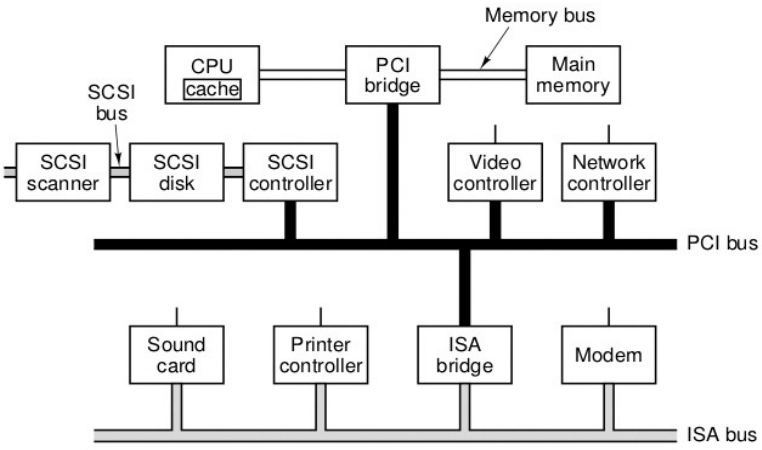
Controllers

- each I/O device has a controller;
- some of the controllers (some video controllers, hard-disk) are located on the I/O boards;
- other controllers (keyboard) are located on the motherboard;
- the job of a controller: control the device and handle bus access for it;
- example – disk controller:
  - when a program wants data from the disk it gives a command to the disk controller,
  - the disk controller issues the seek command – to locate the data on the disk,
  - the drive outputs bit streams – the controller assembles these into words and writes them to the main memory;
- controllers that read/write data from/to the memory are performing Direct Memory Access (DMA):
  - when the transfer is completed, the controller causes an interrupt, the CPU suspends the program it runs and starts an interrupt handler to check for errors, take appropriate actions, etc.
  - when the interrupt handler has finished, the CPU resumes the execution of the program it was running.

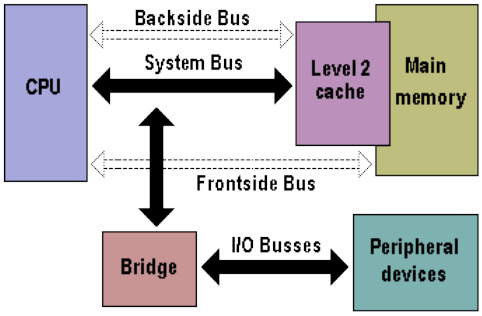
The Bus

- the bus is used both by the CPU and the I/O devices;
- the bus arbiter decides which gets the bus next;
- in general I/O devices have priority, because they cannot usually be stopped (e.g. disks);
- whenever the CPU is alone, it will use the bus;
- when a I/O device is also running, upon request it will be given access to the bus (cycle stealing – slows down the computer);
- this approach worked fine with the early computers, however soon enough the bus could no longer cope with the load (faster CPUs, faster I/O devices);
- example - IBM PC, PS/2 range:
  - new, faster bus, but at the time, a whole industry of I/O devices for the slower bus;
  - the slower bus was ISA (Industry Standard Architecture);
- solution - make computers with 2 or more buses:
  - ISA or EISA (Extended ISA);
  - PCI (Peripheral Component Interconnect) (Intel – public domain, de facto industry standard);
  - USB (Universal Serial Bus) - up to 127 devices.

The Logical Structure of a Personal Computer (revisited)



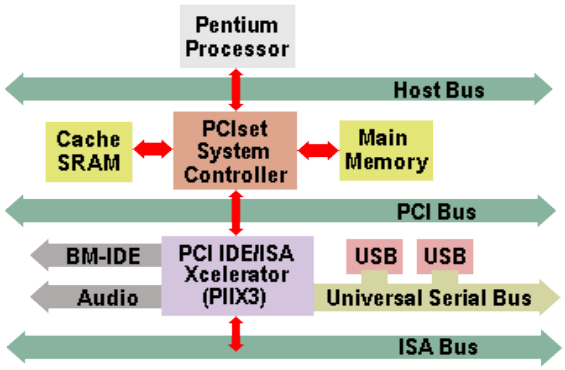
Buses Revisited



- system bus(es): CPU-Memory bus
  - Dual Independent Bus (DIB) architecture:
    - "frontside bus": CPU-Memory-I/O;
    - "backside bus": CPU-Cache;
    - over time, the terms "FSB" and "system bus" came to be used interchangeably;
- I/O buses

USB (Universal Serial Bus)

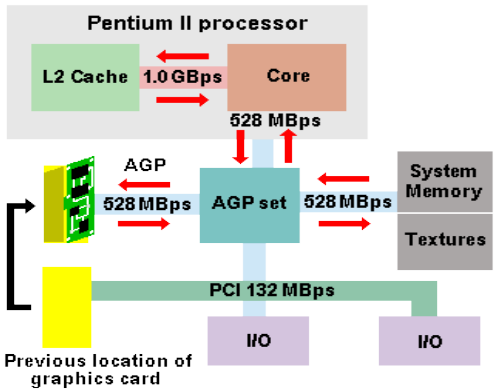
- new standardized connector for attaching I/O devices;
- truely plug-and-play;
- joint development of Compaq, Digital, IBM, Intel, Microsoft, NEC and Northern Telecom;
- USB 1.1 - 1998;
- USB 2.0 - 2000;



Input / Output

AGP – Accelerated Graphics Port

- Remark: bus vs. interface;
- AGP – addressed the need for massive data volumes needed to render 3D graphics;
- a separate connector, operates off the processor bus;
- Direct Memory Execute (DIME) – use system memory as if on the video card;



Input / Output

Buses and Interfaces: Summary (1)

- ISA- Sound cards, modems
  - 2 MB/s to 8.33 MB/s
  - Phased out
- EISA Network, SCSI adapters
  - 33 MB/s
  - Almost entirely phased out; superseded by PCI
- PCIGraphics cards, SCSI adapters, new generation sound cards
  -133 MB/s (standard 32-bit, 33MHz bus)
  -Standard add-in peripheral bus
- AGP Graphics cards
  - 528 MB/s (2x mode), 2 GB/s (8x mode);
  - Standard in all Intel-based PCs from the Pentium II; co-exists with PCI
- USB I/O Devices
  - 1.5-12 MB/s (1.1) – 480 MB/s (2.0);
- IEEE 1394 FireWire I/O Devices
  - 12.5, 25, or 50 MB/s, the cable interface speeds of 100, 200 and 400 MB/s;
- PCI Express: replacement for AGP, PCI
  - up to 8 GB/s

Input / Output

Buses and Interfaces: Summary (2)

PCI 32-bit/33MHz	1064 Mbit/s	**133 MB/s**
PCI 64-bit/33MHz	2128 Mbit/s	**266 MB/s**
PCI 32-bit/66MHz	2128 Mbit/s	**266 MB/s**
AGP 1x	2128 Mbit/s	**266 MB/s**
PCI Express (x1 link)	4000 Mbit/s	**500 MB/s**
AGP 2x	4256 Mbit/s	**532 MB/s**
PCI 64-bit/66MHz	4264 Mbit/s	**533 MB/s**
PCI Express (x2 link)	8000 Mbit/s	**1000 MB/s**
AGP 4x	8512 Mbit/s	**1064 MB/s**
PCI-X 133	8528 Mbit/s	**1066 MB/s**
InfiniBand	10.00 Gbit/s	1.25 GB/s
PCI Express (x4 link)	16.00 Gbit/s	**2 GB/s**
AGP 8x	17.024 Gbit/s	**2.128 GB/s**
PCI-X DDR	17.064 Gbit/s	**2.133 GB/s**
PCI Express (x8 link)	32.00 Gbit/s	**4 GB/s**
HyperTransport (800MHz, 16-pair)	51.2 Gbit/s	**6.4 GB/s**
PCI Express (x16 link)	64 Gbit/s	**8 GB/s**
HyperTransport (1GHz, 16-pair)	64.0 Gbit/s	**8.0 GB/s**

INPUT / OUTPUT DEVICES

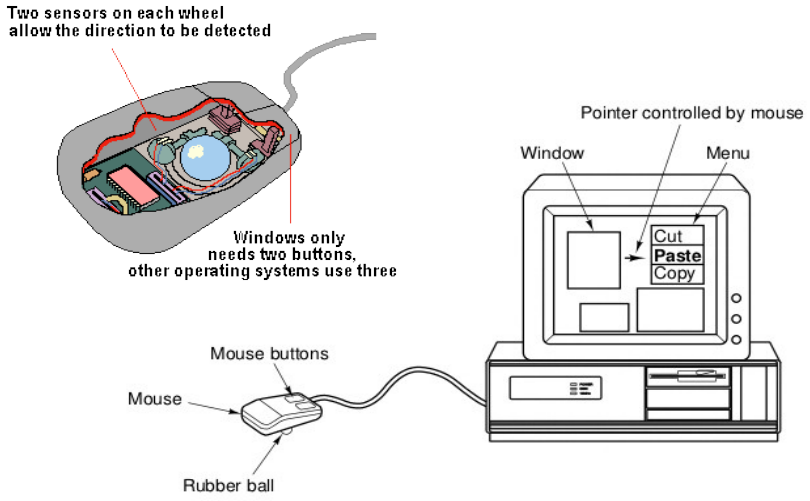
User Input Devices: Keyboards

- computer keyboard = an array of switches, each of which sends the PC a unique signal when pressed;
- types: mechanical (spring loaded) and rubber membrane;
- on PCs: key pressed -> interrupt -> keyboard interrupt handler (part of OS);
  - key released -> another interrupt;
  - key combinations (CTRL-ALT-DEL);
- increasingly complex, with programmable buttons, etc.

User Input Devices: Mice

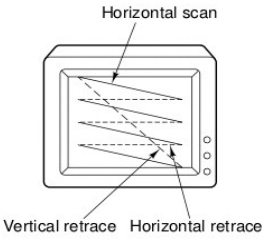
- mouse: I/O device designed to interact with graphical user interfaces (GUIs);
- a plastic box – sits on the table next to the keyboard;
- its moves reflect the moves of a pointer on the screen;
- 1, 2, 3, more buttons; scroll wheels;
- the mouse sends a sequence of 3 bytes to the computer, with every movement;
- constructive types:
  - mechanical,
  - optomechanical,
  - optical.

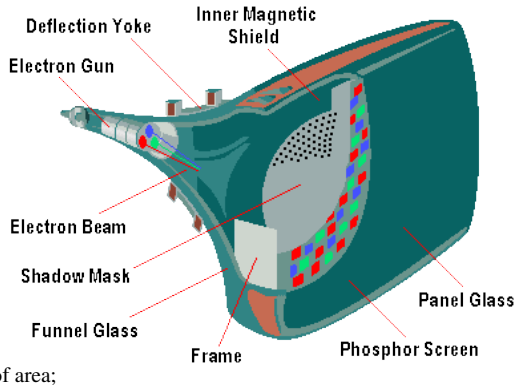
User Input Devices: Mice (continued)



CRT Monitors

- Catode Ray Tube Monitor:
  - electron guns (R, G, B);
  - pixel: 3 (R, G, B) phosphor dots;
  - horizontal scan;
  - vertical scan;





- resolution: the number of pixels per unit of area;
- refresh rate: number of frames displayed per second;

VSF = HSF / number of horizontal lines x 0.95

VSF = vertical scanning frequency (refresh rate) and HSF = horizontal scanning frequency (kHz)

- flicker-free images require at least 75 Hz refresh rates.

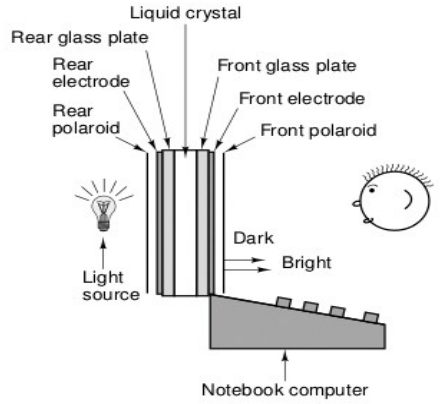
CRT Monitors (continued)

- disadvantages of CRTs:
  - they're heavy and bulky;
  - they're power hungry - typically 150W for a 17in monitor;
  - their high-voltage electric field, high- and low frequency magnetic fields and x-ray radiation have proven to be harmful to humans in the past;
  - the scanning technology they employ makes flickering unavoidable, causing eye strain and fatigue;
  - their susceptibility to electro-magnetic fields makes them vulnerable in military environments;
  - their surface is often either spherical or cylindrical, with the result that straight lines do not appear straight at the edges.

Flat Panel Displays

- liquid crystals = almost transparent substances, exhibiting the properties of both solid and liquid matter. Light passing through liquid crystals follows the alignment of the molecules that make them up - a property of solid matter. In the 1960s it was discovered that charging liquid crystals with electricity changed their molecular alignment, and consequently the way light passed through them - a property of liquids;
- Liquid Crystal Display (LCD): a display technology that relies on polarising filters and liquid crystal cells rather than phosphors illuminated by electron beams to produce an on-screen image:
  - low-cost, dual-scan twisted nematic (DSTN)
  - high image quality thin film transistor (TFT).
- LCD display screens:
  - laptops;
  - PDAs;
  - increasingly more desktop monitors.

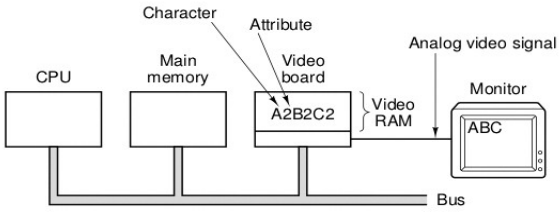
Flat Panel Displays (continued)



Input / Output

Terminals

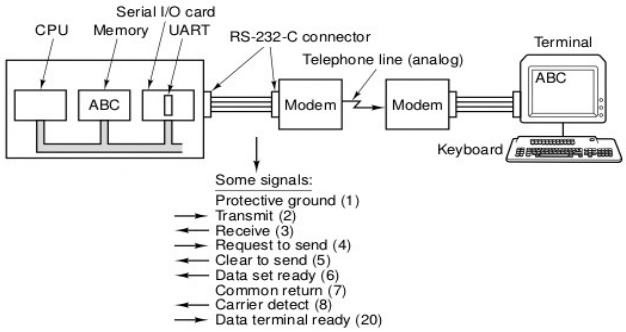
- character-map terminals:
  - hold characters + attributes (color, intensity, etc.) in the video memory



- bitmap terminals:
  - hold pixels in the video memory:
  - resolution 640x480 (VGA), 800x600 (SVGA), 1024x768(XVGA), 1280x960, 1280x1024...
  - color palette: 256, ...
  - specialized hardware on the video controller for scrolling, other video manipulation.

Terminals (continued)

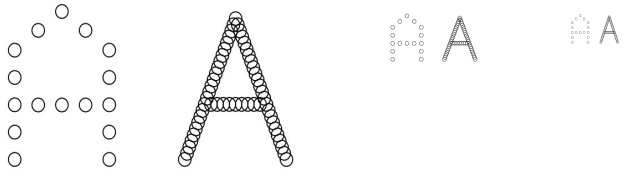
• RS-232-C Terminals:
- terminals that are not directly connected with the computer (esp. for mainframes);
- RS-232-C – a standard developed by Electronics Industries Associations (EIA);



- UART: Universal Asynchronous Receiver Transmitter.

Monochrome Printers

• matrix printers:
- cheap, reliable but poor in graphics, slow, noisy;
- industrial use: supermarket, ATM receipts;
- 7-24 electromagnetically activable needles across a printline;
- increase quality by increasing the number of needles, overlapping dots (see fig.):

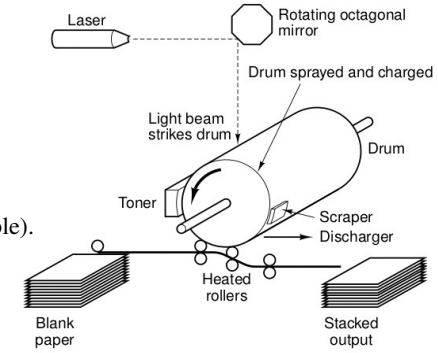


• inkjet printers:
- low-cost(*) home printing;
- a movable print head containing a print cartridge sprays ink through tiny nozzles onto paper;
- at each nozzle, ink droplets are electrically boiled until they explode on the paper, then each nozzle is cooled, the void thus created sucking in the next ink droplet;
- print resolution: 300-600-720-1440 dpi
- cheap, good quality, but slow, with expensive ink cartridges, ink-soaked output.

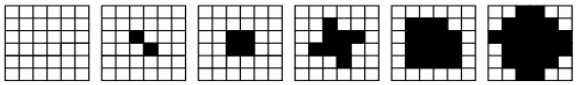
Monochrome Printers (continued)

• laser printers:
- moderate cost, reliable, fast, high image quality;
- same technology as copy machines;
- printing languages – printing engines:
  Adobe Postscript;
  HP PCL;

Printer:
- printing engine,
- CPU,
- memory (fonts – built-in, downloadable).



• printing techniques: halftoning
- image broken in 6x6 pixel cells;
- gray values: 0-255;
- 37 gray zones;



Color Printers

• colors:
- transmitted light images: CRT (RGB);
- reflected light images: printed (photos, glossy pictures) CYMK:
  - cyan (all red absorbed),
  - yellow (all blue absorbed),
  - magenta (all green absorbed),
  - black (to fix imperfections);

• gamut: the complete set of colors that a display or printer can produce;
• no gamut matches the real world:
- limited color sets (16,777,216 discrete colors),
- imperfections in technologies reduce the number further,
- colors not always uniformly distributed in the color spectrum,
- human perception is not uniform (rods and cons in the retina);

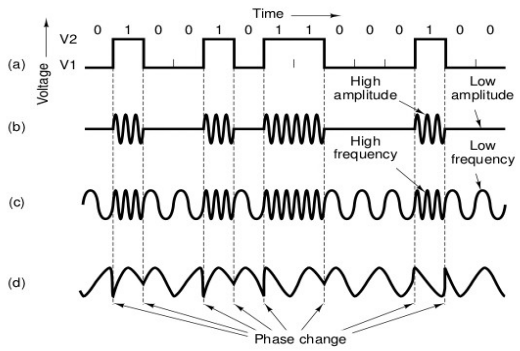
• RGB-CYMK (monitor-printer) transformation is not straightforward:
- pixels on the monitor have 256 intensities, printers must halftone;
- monitors have a dark background, paper is white;
- RGB and CYMK gamuts are different.

Color Printers (continued)

- all types of color printers use CYMK;
- color ink printers:
  - 4 print cartridges (printers cheap, cartridges very expensive),
  - two types of ink: (1) dye-based [bright colors, but they fade in UV light] and (2) pigment-based [do not fade, but not so bright and tend to clog the nozzles];
- solid ink printers:
  - 4 solid blocks of waxy ink, that are melted, then pressed into the paper,
  - up to 10 minutes for a printer to melt the wax;
- color laser printers:
  - 4 toners,
  - require a lot of memory (55 MB for 1200 dpi image, for a 80 square inch image),
  - expensive, but fast printing, lasting colors;
- wax printers:
  - wide ribbons of four-color wax (page size),
  - thousands of heating elements, to melt the wax which is then pressed on the paper;
  - used to be the main color printing technology;
- dye sublimation printers:
  - sublimation: solid changing into gas without going through liquid;
  - heating element (256 temperatures), can produce almost continuous colors;
  - no halftoning, high quality images (on special paper).

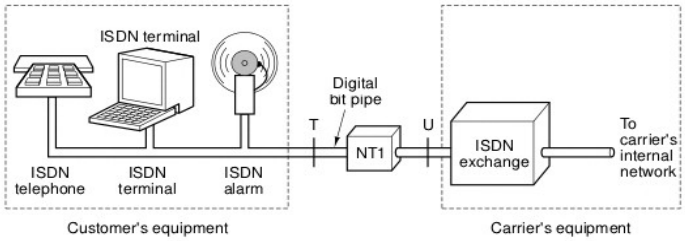
Modems

- I/O devices used to transmit data over telephone lines;
- modulation: a technique for transmitting data over telephone lines:
  (a) the digital signal,
  (b) amplitude modulation,
  (c) frequency modulation,
  (d) phase modulation
  of the carrier wave;
- data is transmitted serially, as streams of bits;
- 8 bits character+start/stop bit = 10 bits;
- 28,800-57,600 bits/sec;
- full-duplex: transmit and receive in the same time (different frequencies);
- half-duplex;
- simplex;
- improving performance: using different modulations at the same time;



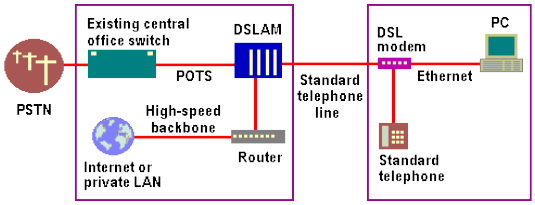
ISDN

- early 1980's: a standard for digital telephony: ISDN (International Services Digital Network) proposed by European PTTs (Post, Telegraph, Telephone Administrations);
- WWW – the killer application;
- 2 independent digital channels, 64,000 bits/sec each, plus a single channel, 16,000 bits/sec; equipment at both ends combine the 3 channels into a 144,000 bps digital channel;
- an ISDN connection for home use:



ADSL

- technologies developed to offer phone companies a way into the cable TV business: xDSL (Digital Subscriber Line);
- very fast(download speeds up to 52 Mbit/s, upload speeds from 64 Kbit/s to over 2 Mbit/s)
- variants:
  - asymmetric (ADSL),
  - high-bit rate (HDSL),
  - single-line (SDSL),
  - very-high-data-rate (HDSL);
- many constructive variants for ADSL modems (USB, Ethernet);



Character Codes: ASCII

- ASCII (American Standard Code for Information Exchange): codes characters on 7 bits (128 characters);

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.asciitable.com

Input / Output

Character Codes: ASCII (Continued)

- Extended ASCII: to address the need for more (e.g. national) characters:

128	Ç	144	É	161	í	177	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	163	ú	179	│	195	├	211	╙	227	π	243	≤
131	â	147	ô	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
134	å	150	û	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
136	ê	152	_	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	·
137	ë	153	Ö	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	171	½	187	╗	203	╦	219	▉	235	δ	251	√
139	ï	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	_
140	î	157	¥	173	¡	189	╜	205	═	221	▌	237	ϕ	253	²
141	ì	158	_	174	«	190	╛	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	
143	Å	160	á	176	▒	192	└	208	╨	224	α	240	≡		

Source: www.asciitable.com

Character Codes: UNICODE

- the basic idea of UNICODE is to assign each character and symbol a 16 bit value, code point;
- each major alphabet has a sequence of consecutive zones in UNICODE:
  - Latin (336), Greek (144), Cyrillic (256), Armenian (96), Devanagari (128), Gurmukhi (128), Oryia (128), Telugu (128), Kannada (128),
  - diacritical marks (112), punctuation marks (128), subscripts, superscripts (48), currency (48), math symbols (256), geometric shapes (96), dingbats (192),
  - symbols for Chinese, Japanese, Korean: 1024 phonetic symbols (katakana, bopomofo), unified Han ideographs (20,992) used in Chinese and Japanese, Korean Hangul syllables (11,156),
  - 6400 characters for local use;
- there are still some problems:
  - 50,000 kanji in a full Japanese dictionary (excluding names),
  - demand for at least 20,000 new entries from the Chinese,
  - braille.

Input / Output

THE REST...

- NETWORKING: COMPUTER NETWORKS course;
- VARIOUS OTHER I/O DEVICES:

  Lab Sessions – Students Presentations.

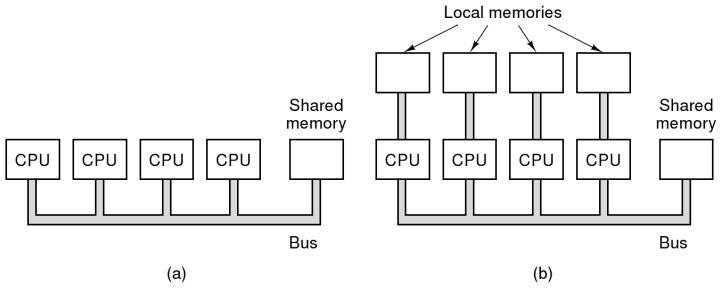


Figure 17: (a) A single bus multiprocessor. (b) A multicomputer with local memories.

3 The Digital Logic Level

In this Section

- Study the basic building blocks computers are made of (gates).
- Look at a special two-valued algebra (Boolean algebra) used to study these basic components.
- Examine fundamental circuits, obtained by combining gates (including circuits for doing arithmetic).
- Examine how gates can be used to store information (memories).
- Examine the CPU, CPU-peripherals interfaces.

3.1 Gates and Boolean Algebra

Gates

- Digital circuit: one of two values are present:
  - 0-1 volt represents one value (e.g. binary 0),
  - 2-5 volt represents the other value (e.g. binary 1).
- Gates are tiny electronic devices that can compute functions of these 2 valued signals.
- Gates are the basic building blocks of computers, and are made of transistors.
- Transistors are electronic devices that act as very fast binary switches.
- A transistor has 3 connections to the outside world:
  - the collector,
  - the base,
  - the emitter.
- Figure 18 illustrates a transistor, and how transistors can be combined to form basic gates.
- Gates are obtain from combinations of transistors and can be seen as functions of input, as soon as the representation of the digital values is decided (e.g. "high" - V_{cc}) is logical 1, and "low" - ground is logical 0). In Figure 18 (b), (c) assumes the choice mentioned here.
- The basic gates and their behavior as functions of their inputs are represented in Figure 19.

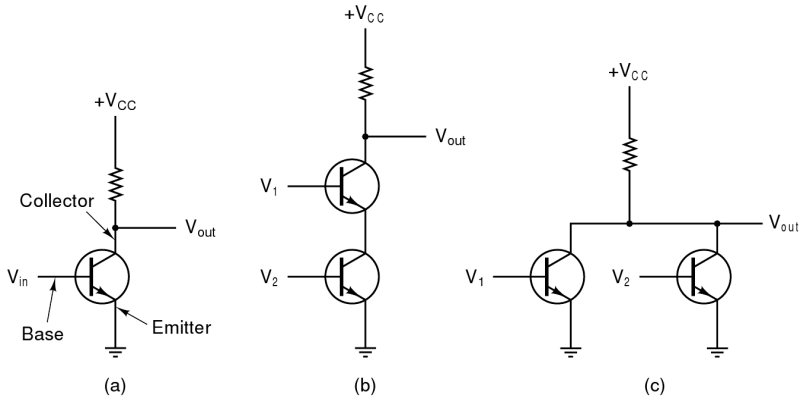
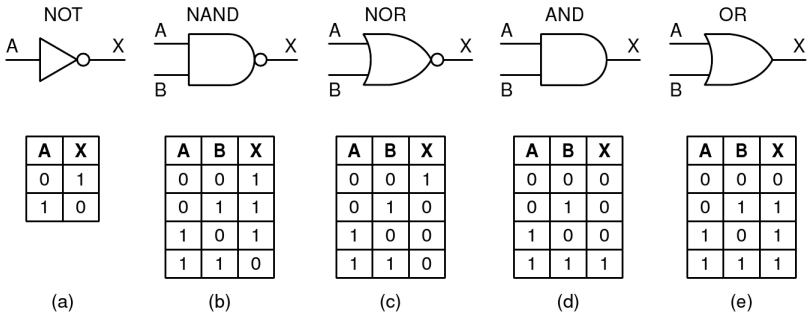


Figure 18: (a) A transistor as an inverter and two transistors combined to form (b) a NAND gate and (c) a NOR gate.



A	X
0	1
1	0

(a)

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

(b)

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

(c)

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

(d)

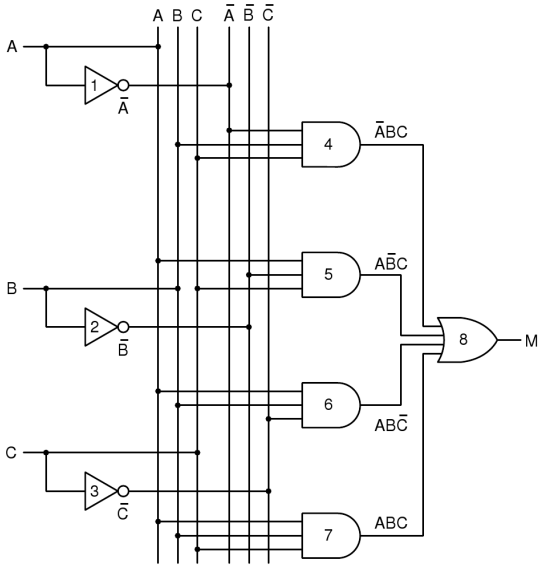
A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

(e)

Figure 19: The basic gates and their behavior.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a)



(b)

Figure 20: (a) The majority function as a truth table and (b) the corresponding circuit.

- Since the AND and OR gates each need 3 transistors, the NAND and NOR gates are used instead in practice.
- Constructive technologies:
  - bipolar: TTL (Transistor-Transistor Logic), ECL (Emitter-Coupled Logic) [high speed],
  - MOS (Metal Oxide Semiconductor) [slower but less power hungry]: PMOS, NMOS, CMOS (used in memories).

Boolean algebra

- Boolean algebra is used to describe the circuits that can be built by combining gates.
- It "captures the essence" of the logical operations AND, OR, NOT.
- Boolean functions are represented by truth tables.
- The majority function $M = f(A, B, C)$, i.e. the ternary function that returns the value that appears as an input the most times, is represented in Figure 20.

- The truth table representation of boolean functions may become cumbersome with the increase in the number of inputs.
- To make things easier, one can use the notation for multiplication (AND) and addition (OR) and overbar for negation.
- The compact representation of the majority function:

$$M = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC$$

- The compact representation records the configuration of inputs for output 1.
- A function of n variables can be described by giving a "sum" of at most 2^n "product" terms (DNF).
- This formulation leads to a straightforward way to implement a boolean function using standard gates.
- However, many times there are more efficient ways to implement digital circuits.

Implementation of Boolean Functions

- Some representation conventions: whenever lines cross, no connection is implied unless a heavy dot marks the connection.
- Given a boolean function, its implementation is done in the following way:
  1. Write down the truth table for the function.
  2. Provide inverters to generate the complement of each input.
  3. Draw an AND gate for each term with a 1 in the result column (from the truth table).
  4. Wire the AND gates to the appropriate inputs.
  5. Feed the output of all the AND gates into an OR gate.
- It is simpler to construct circuits that have fewer, simpler gates:
  - Replace multi-input gates with two-input gates: $A+B+C+D$ with $(A+B)+(C+D)$.
  - Replace various types of gates with only one type (this is possible, see Figure 21, {NAND} and {NOR} are complete sets of boolean connectives).

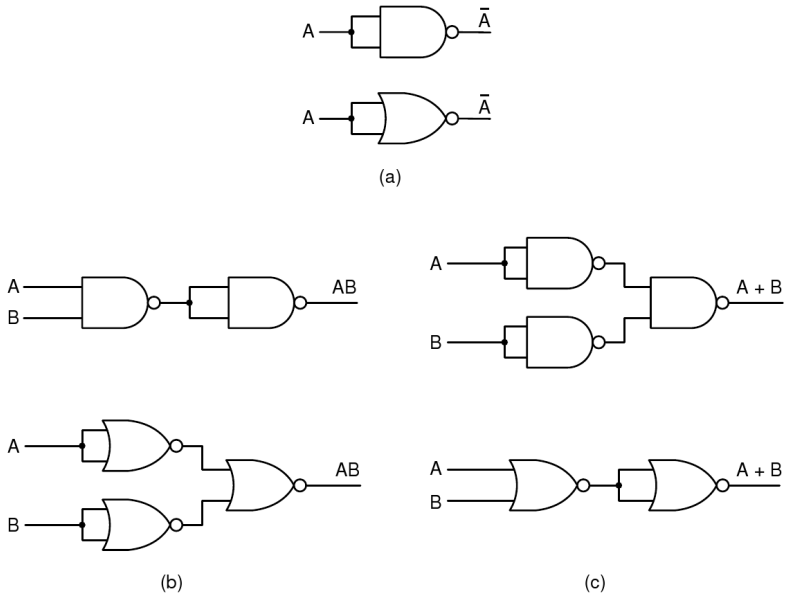
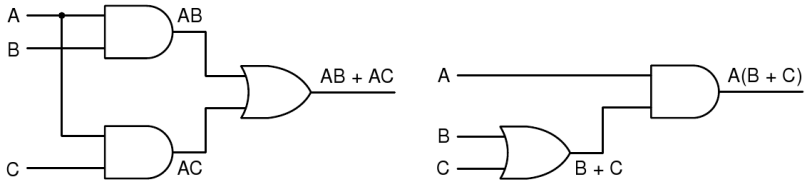


Figure 21: The basic gates represented in terms of NAND, NOR: (a) NOT, (b) AND and (c) OR.



A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

(a)

A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

(b)

Figure 22: Two equivalent Boolean functions.

Name	AND form	OR form
Identity law	$1A = A$	$0 + A = A$
Null law	$0A = 0$	$1 + A = 1$
Idempotent law	$AA = A$	$A + A = A$
Inverse law	$A\overline{A} = 0$	$A + \overline{A} = 1$
Commutative law	$AB = BA$	$A + B = B + A$
Associative law	$(AB)C = A(BC)$	$(A + B) + C = A + (B + C)$
Distributive law	$A + BC = (A + B)(A + C)$	$A(B + C) = AB + AC$
Absorption law	$A(A + B) = A$	$A + AB = A$
De Morgan's law	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$

Figure 23: Some laws of Boolean algebra.

Circuit equivalence

- Circuit equivalence is exploited to reduce the number of gates and simplify a circuit.
- Start with a Boolean function and apply the laws of Boolean algebra to simplify it, see Figure 22.
- The laws of boolean algebra are illustrated in Figure 23
- Figure 24 shows alternative notations for different gates.
- Figure 25 shows equivalent representations of the implementation of exclusive OR (XOR).

Representation conventions and digital devices

- According to the assignment of 0,1 to voltages (e.g. 0V, 5V respectively) in digital devices, distinguish:
  - positive logic: 0 for 0V, 1 for 5V,
  - negative logic: 1 for 0V, 0 for 5V.
- Figure 26 shows how the same device can implement different gates with different choices of representation.

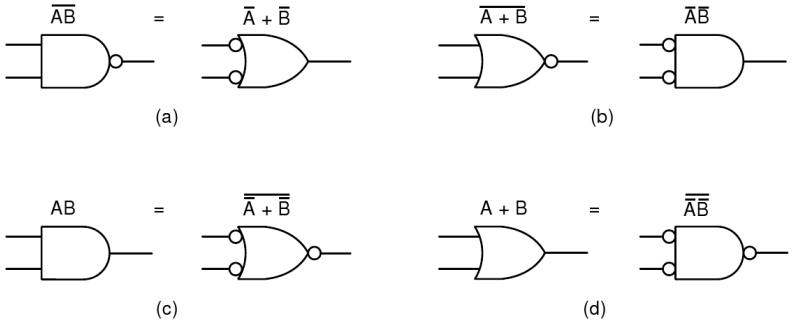


Figure 24: Some alternative notations for (a) NAND, (b) NOR, (c) AND, (d) OR.

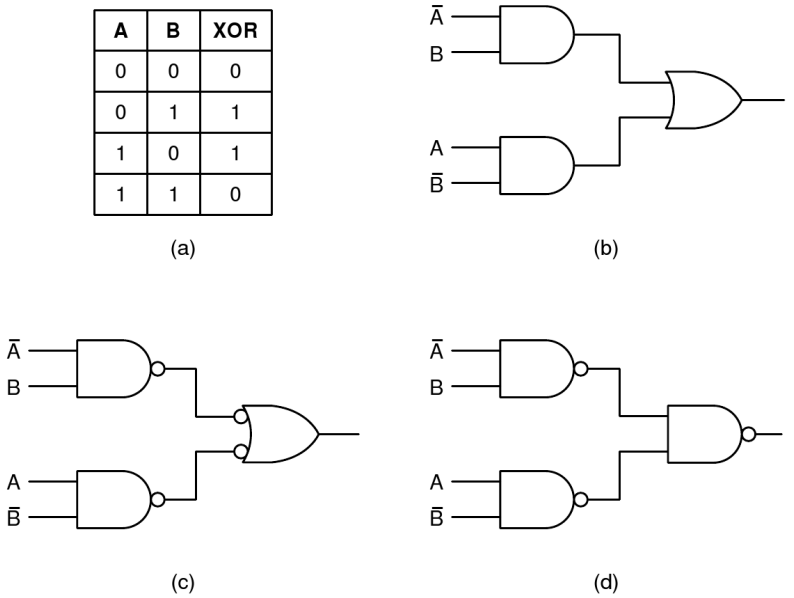


Figure 25: (a) XOR and some of its equivalent implementations (b), (c), (d).

A	B	F
0^V	0^V	0^V
0^V	5^V	0^V
5^V	0^V	0^V
5^V	5^V	5^V

(a)

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

(b)

A	B	F
1	1	1
1	0	1
0	1	1
0	0	0

(c)

Figure 26: (a) Digital device in (b) positive logic and (c) negative logic.

3.2 Basic Digital Logic Circuits

Integrated Circuits

- Gates are packed on units called Integrated Circuits - ICs or chips.
- ICs: a number of gates on a plaque, pins.
- Classification of ICs:
  - (Small Scale Integration): 1 to 10 gates,
  - MSI (Medium Scale Integration): 10 to 100 gates,
  - LSI (Large Scale Integration): 100 to 100,000 gates.
  - VLSI (Very Large Scale Integration): > 100,000 gates.
- Figure 27 shows a SSI chip with 4 gates, 14 pins (12 + power and ground connectors) and a notch to ensure correct orientation.
- A gate delay of 1-10 nsec occurs.
- Implementation issues:
  - A circuit with 5,000,000 NANDs would need 15,000,002 pins x 0.1 inch, i.e. approx 18km long chip.
  - Solution: design circuits with a high gate/pin ratio.

Combinatorial circuits - Multiplexers

- Combinatorial circuits are circuits with multiple inputs and multiple outputs, where the outputs are uniquely determined by the current inputs.
- A multiplexer is a circuit with 2^n data inputs, n control inputs and 1 output, see Figure 28.
- Multiplexers can be used for parallel-to-serial converters.
- Demultiplexers are the inverse circuits.
- Figure 29 shows the use of a multiplexer to implement the majority function.

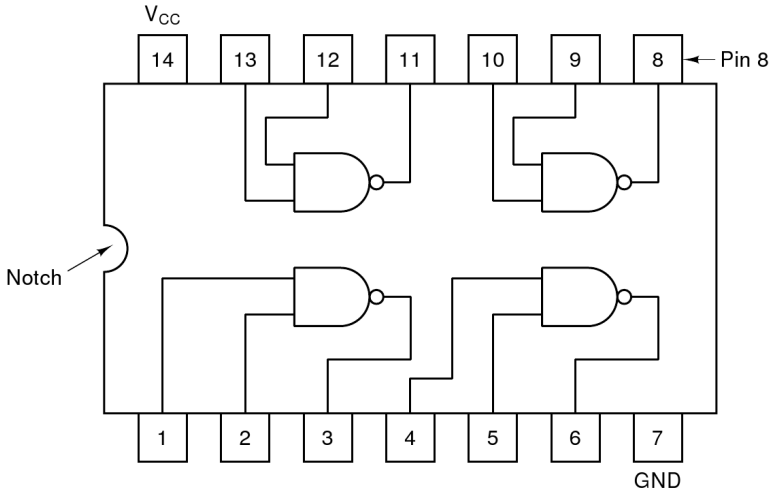


Figure 27: A SSI chip.

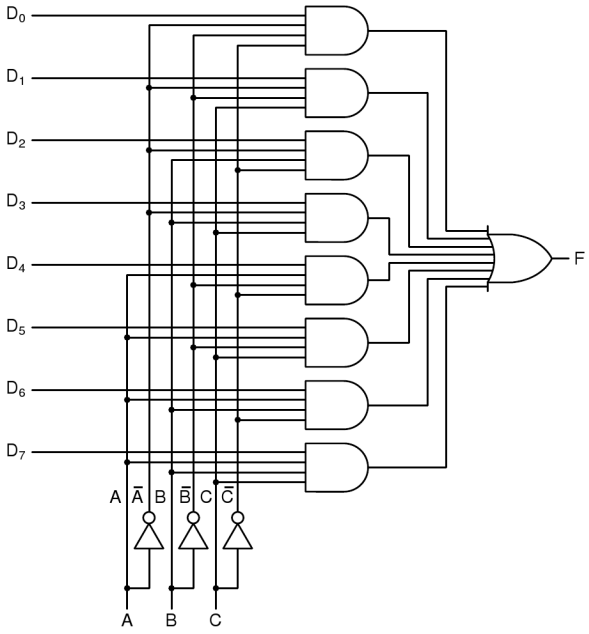


Figure 28: A multiplexer.

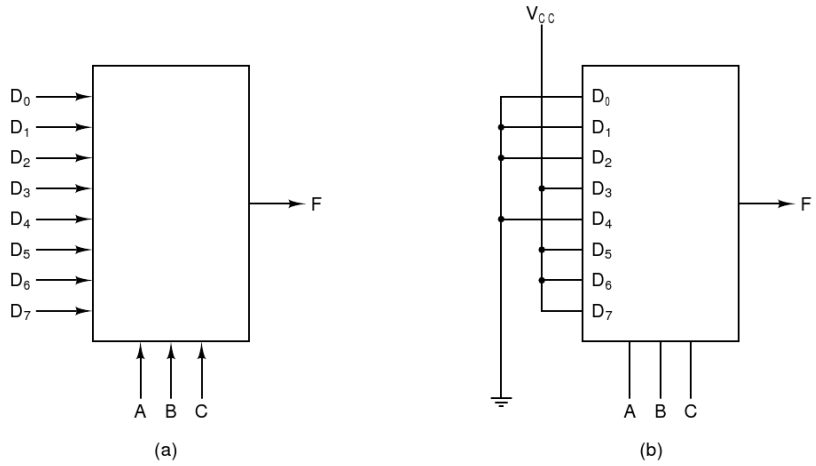


Figure 29: (a) A MSI multiplexer, (b) used to implement the majority function.

Combinatorial circuits - Decoders

- Decoders have n inputs and 2^n outputs - the number formed by the input is used to select (i.e. set to 1) exactly one of the 2^n output lines. See Figure 30.
- Example of use: memory addressing.

Combinatorial circuits - Comparators

- A comparator circuit compares two words, see Figure 31.

Combinatorial circuits - Programmable Logic Arrays

- Boolean functions can be written as "sums" of "products".
- Programmable Logic Arrays (Pleas) (see Figure 32) are circuits that can be used to implement arbitrary boolean functions by providing:
  - n input lines ($2n$ internally by also providing their negations),
  - m AND gates each with inputs a subset of the input lines,
  - a $n \times m$ matrix of fuses that specify which input goes into the ANDs,
  - p OR gates that take as input outputs of the m AND gates,
  - a matrix of $m \times p$ fuses to specify which AND output goes into the OR input,
- The circuit is programmed by blowing some of the fuses.

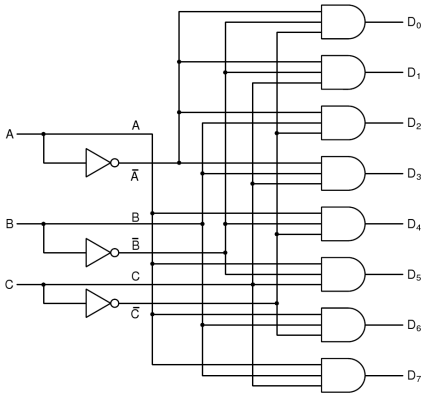


Figure 30: A 3 to 8 decoder.

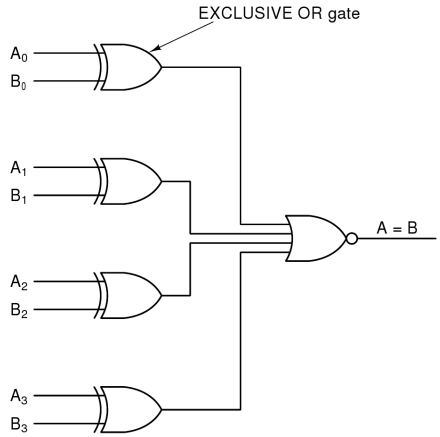


Figure 31: A comparator.

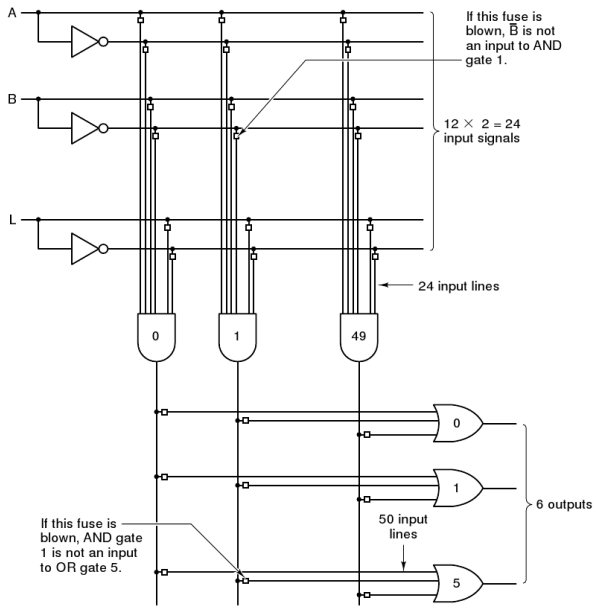


Figure 32: A 12 input 6 output programmable logic array.

Arithmetic circuits - Shifters

- Shifters have n inputs, n outputs and a control line (controlling the direction of the shift - 0 for left and 1 for right), see Figure 33.

Arithmetic circuits - Adders

- A half adder computes the sum and carry of two bits, see Figure 34.
- A full adder also takes into account the carry bit from the right (carry in), see Figure 35.
- Full bit adders can be put together to form n bit adders.
- This type of adders is known as ripple carry adders.
- Ripple carry adders have disadvantages - delay.
- Improvement - carry select adder:
  - divide a $2n$ bit adder into an n bit lower half and an n bit upper half,

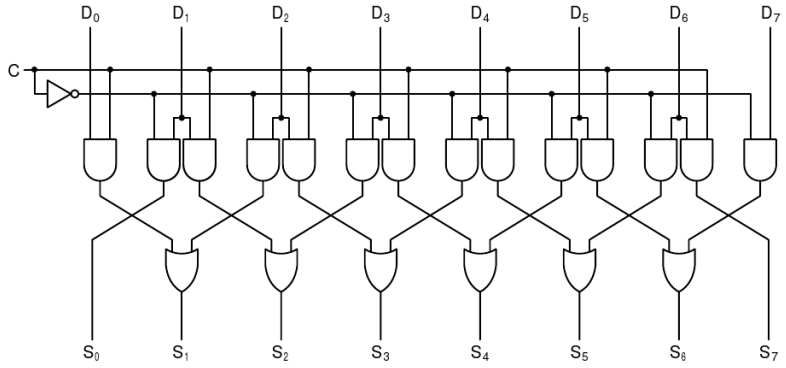


Figure 33: An 8 bit shifter.

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

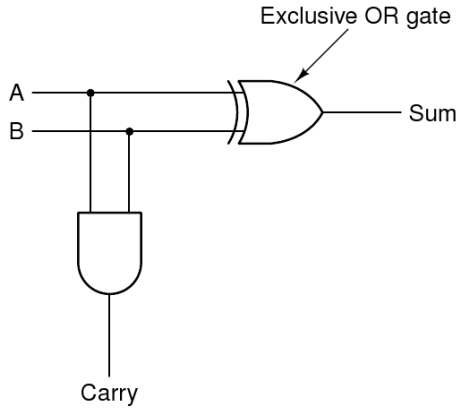
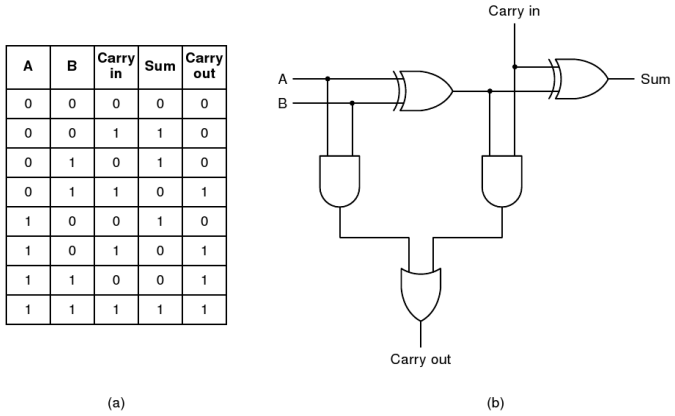


Figure 34: A half adder.



A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 35: A full adder (a) specification and (b) circuit.

- duplicate the upper half hardware,
- one of the upper halves gets 0 as a right carry, the other one gets 1 as a right carry,
- the two upper halves run in parallel,
- for the final result, one of the two circuits is selected (according to the result of the carry from the lower half).
- 100% increase in speed at the cost of 50% more hardware.

Arithmetic Circuits - Arithmetic Logic Units

- Arithmetic Logic Units (ALU) are circuit that perform basic word operations: AND, OR, NOT, sum.
- Figure 36 illustrates a 1 bit ALU. The inputs F_0, F_1 control the operations to be performed, ENA, ENB enable inputs, INVA forces $\overline{A}$.
- n bit ALUs are obtained by combining n 1 bit ALUs (bit slices), see Figure 37.

Clocks

- Clocks (see Figure 38) are circuits that provide synchronization of devices by emitting series of pulses with a precise width and with precise intervals between pulses (clock cycle time).
- The clock is usually controlled by a crystal oscillator.

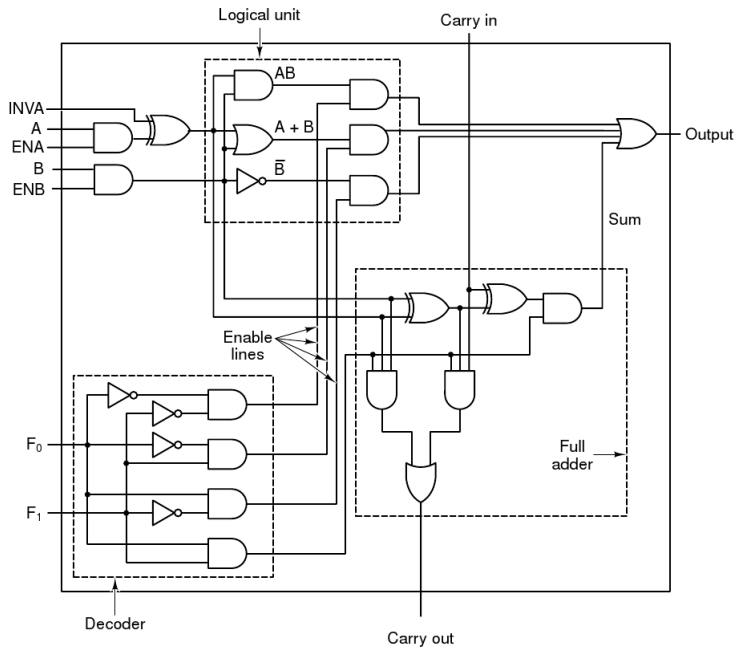


Figure 36: 1 bit ALU.

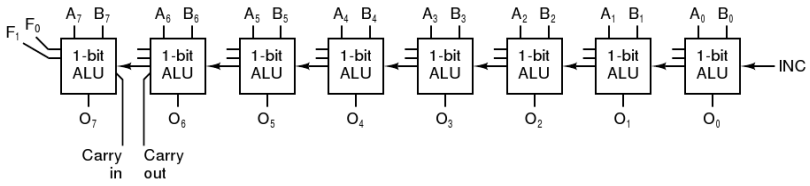


Figure 37: 8 bit ALU.

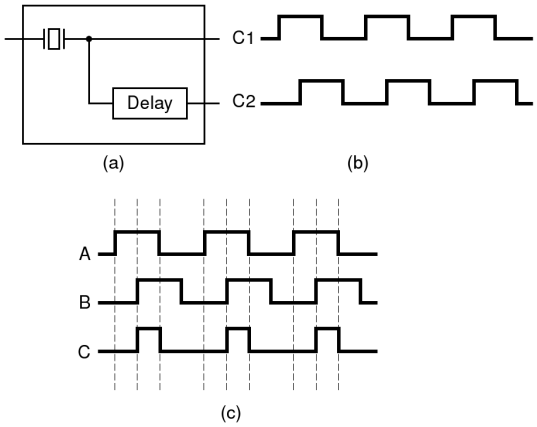


Figure 38: Clocks: (a) subdivisions of the clock cycle by inserting a delay, getting 4 references (b) rising edge of C1/ falling edge of C1/rising edge of C2/falling edge of C2 and (c) asymmetric clock - basic clock shifted and ANDed with the original circuit.

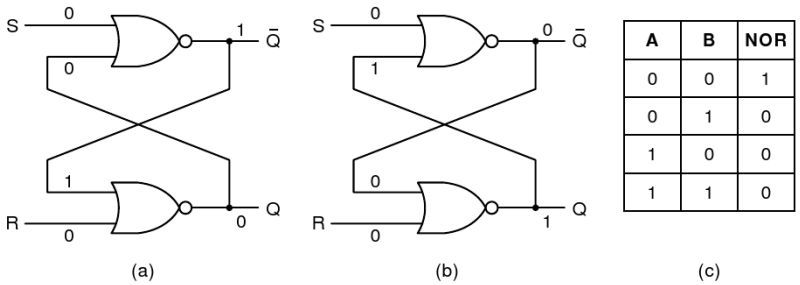
3.3 Memory

Latches

- Latches are circuits that "remember" previous input values.
- A SR latch, see Figure 39 has the following:
  - inputs: S, for setting, R for resetting,
  - outputs, Q, $\overline{\text{Q}}$.
- SR latches are not uniquely determined by the current inputs:
  - S = R = 0 has 2 stable states, depending on the value of Q,
  - S = 1 has the effect Q = 1,
  - R = 1 has the effect Q = 0,
  - the circuit "remembers" whether S or R was last on.

Clocked latches

- Adding a clock (enable, strobe) to the SR latch, prevents it from changing the state, except at specified times, see Figure 40.
- SR latches have a problem: S = R = 1 (the latch jumps to one of the stable states at random).



A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

Figure 39: A NOR SR latch.

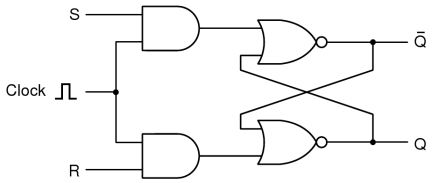


Figure 40: A clocked SR latch.

- Clocked D latches fix the ambiguity of SR latches by avoiding the situation that leads to the ambiguity, see Figure 41.

Flip-flops

- Flip-flops - are circuits that sample the value on a certain line at a particular time and store it.
- The clock transition occurs during the clock transition – from 0 to 1 (the rising edge) or from 1 to 0 (the falling edge), and not on plateaux.

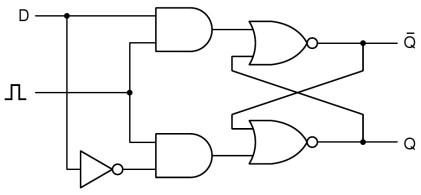


Figure 41: A clocked D latch.

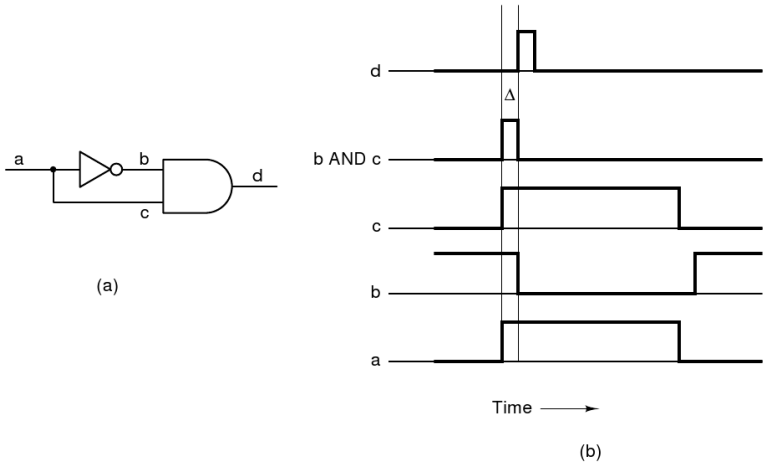


Figure 42: (a) Pulse generator and (b) pulse signal.

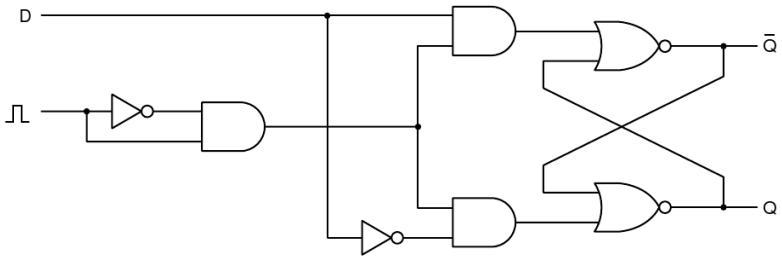


Figure 43: A D flip-flop.

- Flip-flops differ from latches in that they are edge triggered as opposed to level triggered.
- Flip-flops can be obtained by combining a pulse generator (Figure 42) with a latch (Figure 43).
- Figure 44 illustrates the various representations of latches and flip-flops.
- Latches and flip-flops may have additional inputs (Set, *Preset* to force $Q = 1$, *Reset*, *Clear*, to force $Q = 0$)

Registers

- Flip-flops are packed together on ICs to form registers, see Figure 45.

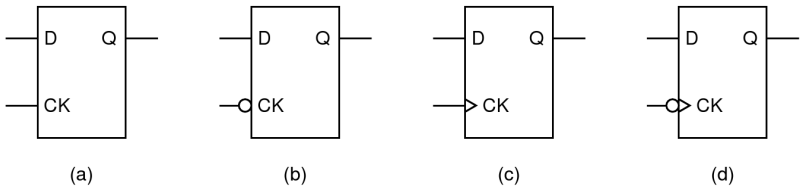


Figure 44: Representations of (a), (b) latches and (c), (d) flip-flops.

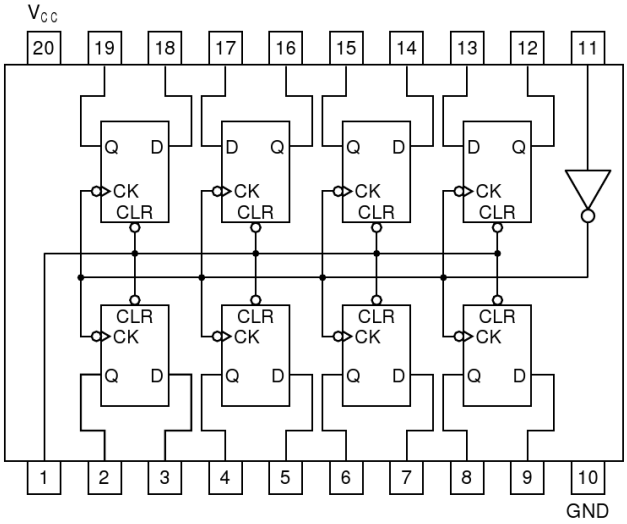


Figure 45: An 8 bit register.

Memory chips

- Memory chips are more complicated than the registers.
- Figure 46 illustrates a memory chip holding 4 3 bit memory words.
- Inputs:
  - 3 data inputs ($I_0 - I_2$),
  - 2 address inputs (A_1, A_2),
  - 3 control inputs: CS (chip select), RD(read or write), OE (output enable).
- Outputs: 3 data outputs ($O_0 - O_2$).
- The chip has 14 pins (as opposed to the 8 bit register before).
- CS selects the memory chip, RD with value 1 (read) or 0 (write).
- Write:
  - $CS * RD * OE$ will be 0, so no output,
  - $CS * \overline{RD}$ together with A_1A_2 will enable one of the write gates,
  - $I_0I_1I_2$ will be written in the flip-flops corresponding to the selected word.
- Read:
  - $CS * RD * OE$ will be 1, so the output is enabled,
  - $CS * \overline{RD}$ will be 0, so will the write gates,
  - A_1A_2 will select the word whose content is sent to the output.

Noninverting buffers

- In practice, the same lines are used for input and output.
- Noninverting buffers, see Figure 47, allow the interruption of cable, such that the memory chip will not attempt to get input and output at the same time.
- Noninverting buffers are tri-state devices (they can output 0, 1 or nothing at all).

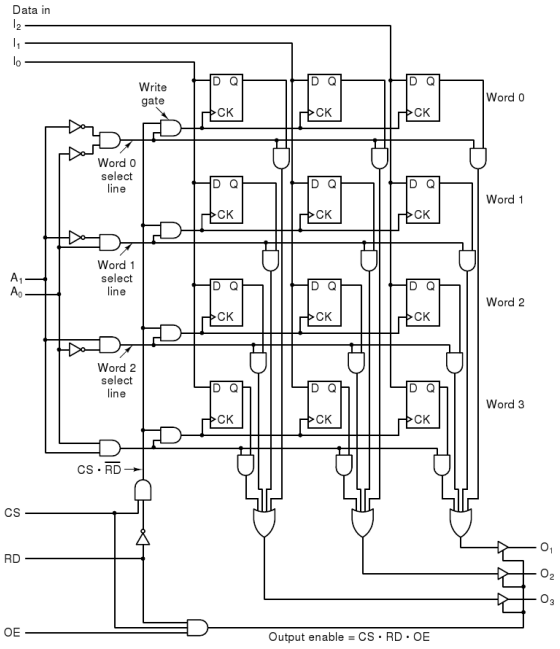


Figure 46: A 4 x 3 bit memory.

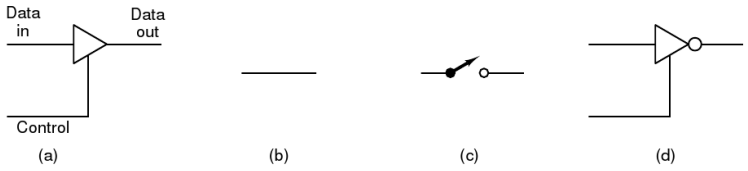


Figure 47: (a) A noninverting buffer (IN: data, control, OUT: data), (b) the effect of the control being 1, (c) the effect of the control being 0, (d) an inverting buffer.

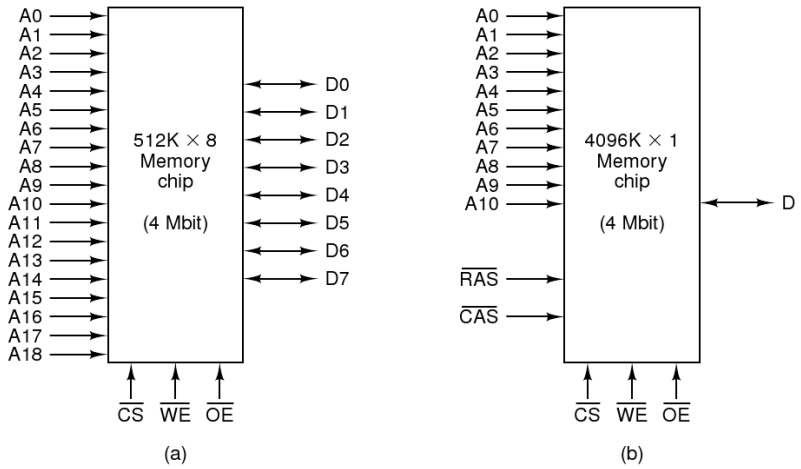


Figure 48: The organization of a 4 Mbit memory chip: (a) 512K x 8, (b) 4096K x 1.

Memory organization

- The previous design can be easily extended.
- For efficiency reasons, the number of words should be powers of 2.
- Memory chips observe Moore's law.
- There are various ways in which to organize chips. In Figure 48 we have 4 Mbit chips organized in different ways: 512K x 8, 4096K x 1 (memories tent to be measured in bits, not in bytes).

- Terminology:
  - asserting - setting a value (positive, if the line is set to 1, negative otherwise),
  - negating - the opposite of asserting (same as disconnecting).
- The 512K x 8 chip:
  - $\overline{\text{CS}}$ (Chip Select) asserted to select the chip.
  - $\overline{\text{WE}}$ (Write Enable) asserted to indicate that data is being written.
  - $\overline{\text{OE}}$ (Output Enable) asserted to drive the output signals.
- The 4096 x 1 chip:
  - internally, it is a 2048 x 2084 matrix of 1 bit cells,

  - the chip writes/reads 1 bit at a time,
  - $\overline{\text{RAS}}$ (Row Address Strobe),
  - $\overline{\text{CAS}}$ (Column Address Strobe).
- The organization of memories as matrices of n x n reduces the number of pins needed, but addressing is slower, since it has to be done twice, once for rows, once fore columns.

RAMs and ROMs (revisited)

- RAMs (Random Access Memories) are memories that can be read and written.
  - Static RAMs:
    - constructed using D flip-flops,
    - keep their contents as long as there is power,
    - very fast (popular as level 2 cache memories).
  - Dynamic RAMs:
    - use transistors and capacitors instead,
    - have to be refreshed every few milliseconds,
    - need less transistors per bit (1+capacitor vs. 6 for a flip-flop),
    - high density but slower,
    - FPM (Fast Page Mode) - matrix of bits,
    - EDO (Extended Data Output) - improve memory bandwidth.
  - S(ynchronous)DRAM is a combination of static and dynamic RAM (used in large caches and main memory) (FPM and EDO are asynchronous).
- ROMs (Read Only Memories)
  - cheaper to build (in large numbers), but the mask may cost,
  - PROM (Programmable ROM) eliminates the cost of the mask, can be written (programmed) once,
  - EPROM (Erasable PROM) can be reused,
  - EEPROM - improvement over PROM - no special device needed to program it.
  - examples of EEPROM - flash memory.
- Observer the transition: ROM $\rightarrow$ NVRAM (Nonvolatile RAM).

3.4 CPU Chips and Buses

CPU Chips

- All modern CPUs are contained on a single chip.
- All interactions to the outside world is done through the processor's pins.
- These pins communicate with memory chips and I/O chips through buses.
- Address pins:
  - the CPU puts a memory address on the address pins to load the word (instruction) from the respective address.
- Data pins:
  - the memory sends the instruction to the data pins,
- Control pins:
  - the CPU also needs arguments, so it informs the memory over the control pins that is wants to read data,
  - the memory tells the CPU over the control lines whether data is available.
  - bus control: the CPU tells the bus what it wants to do (whether it wants to use it),
  - interrupts: input from the I/O devices to the CPU,
  - bus arbitration: needed to regulate traffic on the bus (CPU counts as an I/O device in this context),
  - coprocessor signalling: pins for making and granting requests to/from the coprocessor,
  - status: provide or accept status information,
  - miscellaneous: various, e.g. backward compatibility.
- Other pins: power, ground, clock signal.
- Key parameters for CPU performance:
  - the number of address pins: m can address 2^m addresses (usual values for m are 16, 20, 32, 64).
  - the number of data pins: n can transfer an n bit word at once (usual values for n are 8, 16, 32, 64).
- Figure 49 illustrates a typical microprocessor.

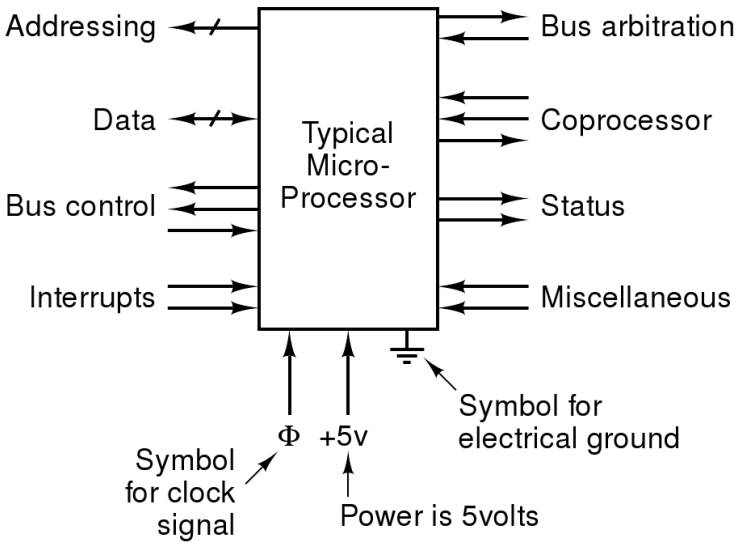


Figure 49: The logical pinout of a generic microprocessor.

Computer buses

- Buses = electrical pathway between multiple devices.
- Types: Internal buses (ALU-registers), external buses (CPU-memory-I/O devices), see Figure 50.
- Bus protocols are sets of well defined rules about how the bus works, and which all devices connected to the bus must obey.
- Examples: Omnibus (PDP-8), Unibus (PDP-11), Multibus (8086), ISA bus (PC/AT), EISA (80386), PCI bus (many PCs), SCSI (PCs, workstations), Nubus (Macintosh), USB (modern PCs), FireWire, VME bus (physics lab equipment), Camac bus (high energy physics).

- Some devices attached to the bus are active and can initiate transfers - masters, whereas some of them are passive - slaves.
- Example:
  - CPU - master, memory - slave: fetching instructions and data,
  - CPU - master, I/O device - slave: initiating data transfer,
  - CPU - master, coprocessor - slave: CPU handling instructions to the coprocessor,
  - I/O - master, memory - slave: DMA.

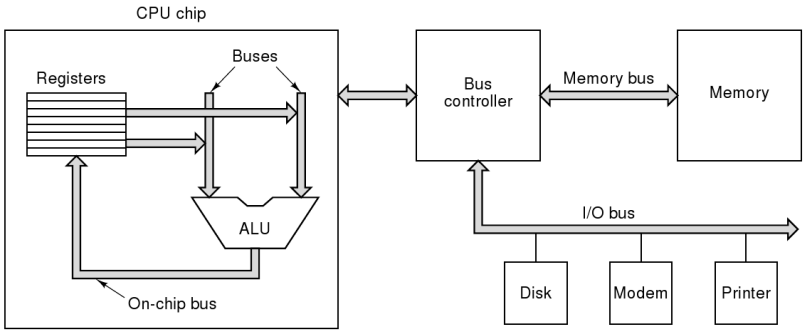


Figure 50: A computer system with multiple buses.

  - coprocessor - master, CPU - slave: coprocessor fetching operands from the CPU.

- Bus drivers - essentially digital amplifiers for bus masters (signals are not powerful enough to power the bus, usually),
- Bus receivers - chips that connect the slaves to the bus.
- Bus transceivers - for devices that can be both masters and slaves.
- A bus, like a CPU, has address, data and control lines, but there is not necessarily a one-to-one mapping between the CPU pins and the bus signals (decoders are places between the CPU and the bus).

Bus design issues

- The more address and data lines, the better.
- However, more lines make the bus more expensive, a tradeoff may be necessary.
- Example: Figure 51 illustrates the evolution of the address buses for Intel processors, backward compatibility leading to a messy design.
- To increase the data transfer through the buses:
  - increase the number of data wires (same design problems mentioned above),
  - decrease the bus cycle time (difficult - bus skew - different signals have different speeds, backward compatibility issues).
  - recent trend: transition to serial buses - high speed and simple design.
- Multiplexed buses use the same wires for both data and address signals (slower bus).

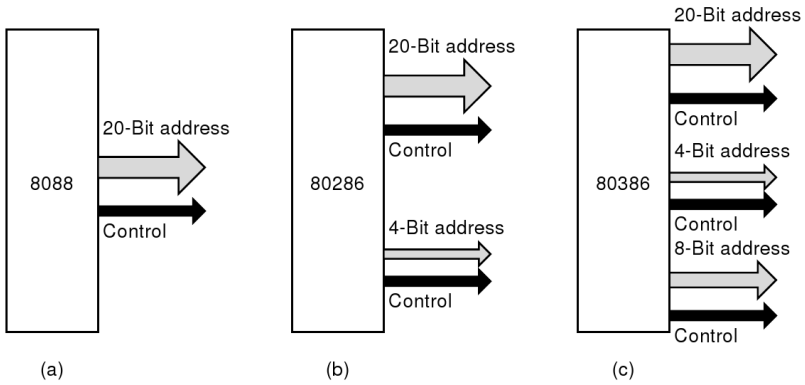


Figure 51: The evolution of an address bus over time (a) enough for addressing 1 Mb, (b) extended to address 16 Mb, (c) extended further to address 1 Gb.

Bus clocking - synchronous buses

- Synchronous buses - driven by a crystal oscillator, all bus activities take an integral number of these bus cycles.
- Figure 52 illustrate the timing of a read operation on a synchronous bus.

Bus clocking - asynchronous buses

- Synchronous buses are geared towards the slowest device in the configuration.
- Asynchronous buses do not have a master clock, but a synchronization signal.
- A full handshake - set of interlocking signals ensure the correctness of operations.
- Figure 53 illustrates a read operation on an asynchronous bus.
- Note that despite the apparent obvious advantage, most of the buses are still synchronous (compatibility, investment, simplicity).

Bus arbitration

- Bus arbitration decides which device gets to use the bus.
- In the case of centralized arbitration, see Figure 54, access is granted by a bus arbiter.

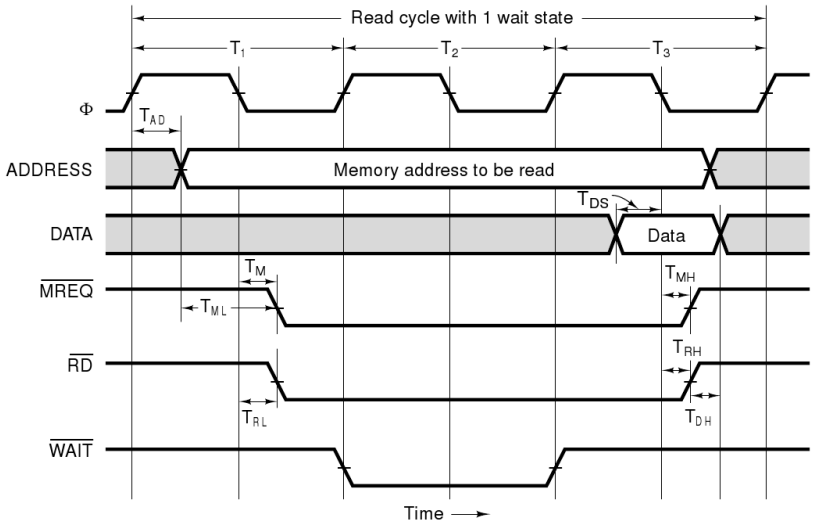


Figure 52: Timing of a read operation on a synchronous bus.

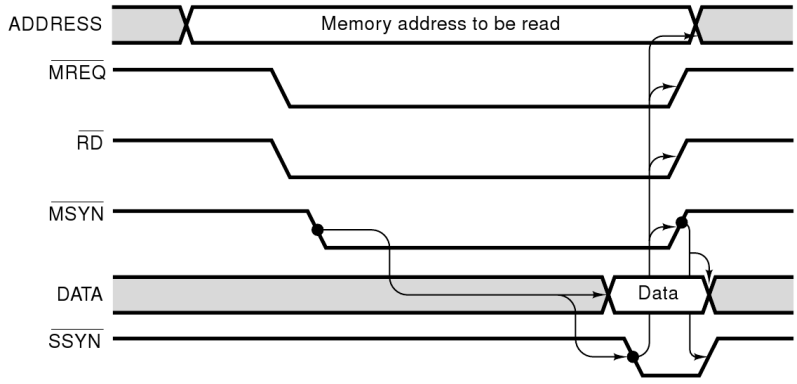


Figure 53: A read operation on an asynchronous bus.

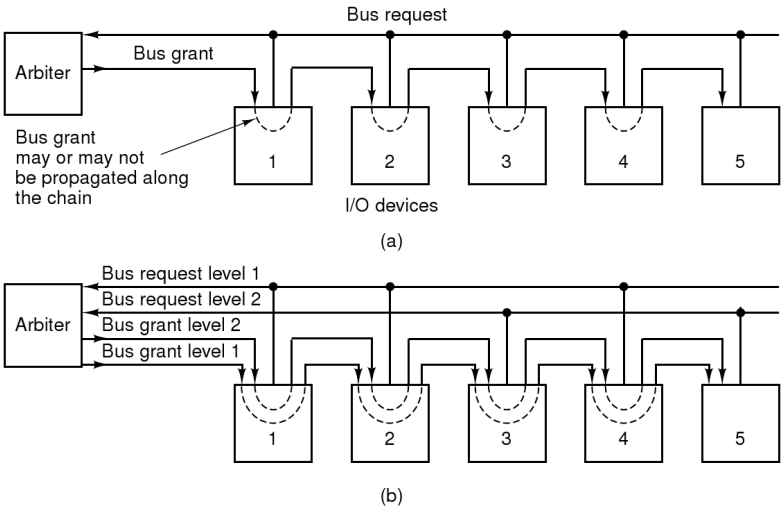


Figure 54: Centralized bus arbitration: (a) with daisy chaining and (b) multi-level daisy chaining.

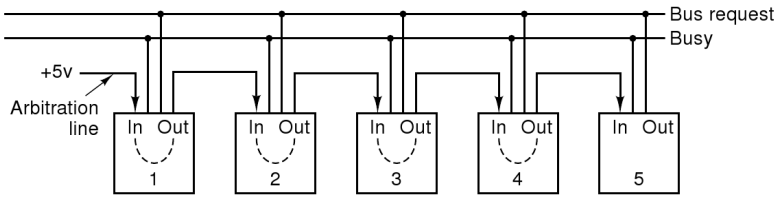


Figure 55: Decentralized bus arbitration.

- Access is granted to the arbiter to the nearest device that requests access daisy chaining.
- There could be several access levels of access priorities.
- In systems with only one bus the CPU gets the lowest priority.

- Decentralized bus arbitration is possible, see Figure 55:
  - prioritized bus request lines (more bus lines, number of devices limited to the number of request lines),
  - decentralized daisy chaining (same behavior as the centralized case, but cheaper and easier to implement).

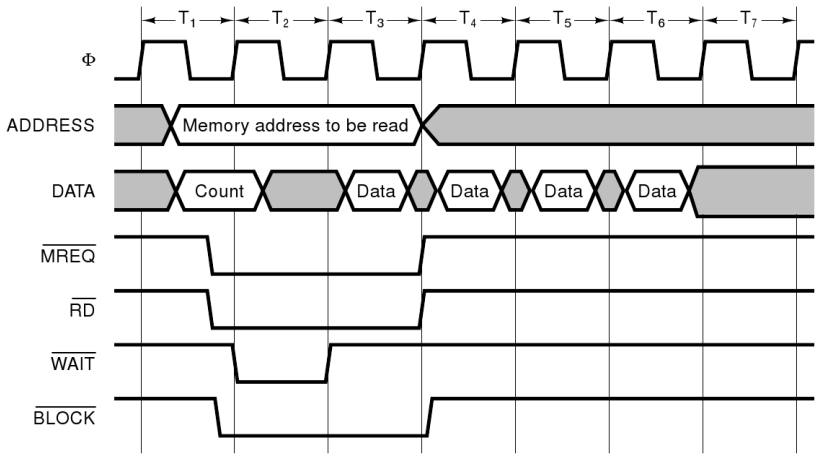


Figure 56: Block operations on buses.

Block operations

- Block read/writes are used in some cases, e.g. when using caching, as illustrated in Figure 56.
- On multiprocessors read-modify-write cycles are used to prevent individual processors modifying critical data structures at the same time. The processor that gets the bus does not release it before the operation was completed.

Handling interrupts

- When the CPU commands an I/O device to do something, it expects back an interrupt.
- Interrupt controllers are chips that arbiter the interrupts.
- Figure 129 illustrates the 8259A controller.
- Up to 8 devices can be connected to the 8259A controller.
- More can be connected using multiple controlled cascaded.
- The procedures for handling interrupts are stored in hardware tables - interrupt vectors.

3.5 Example CPUs

An Intel CPU

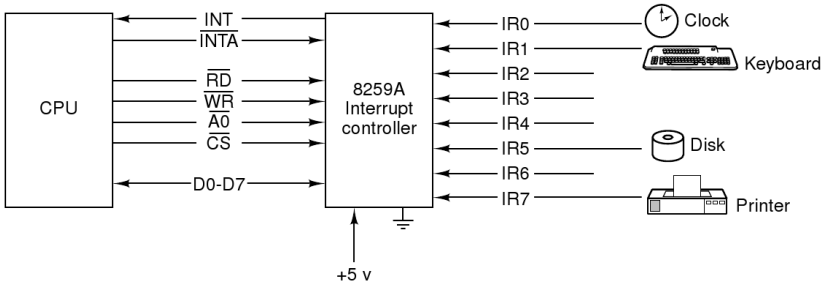


Figure 57: The 8259 interrupt controller.

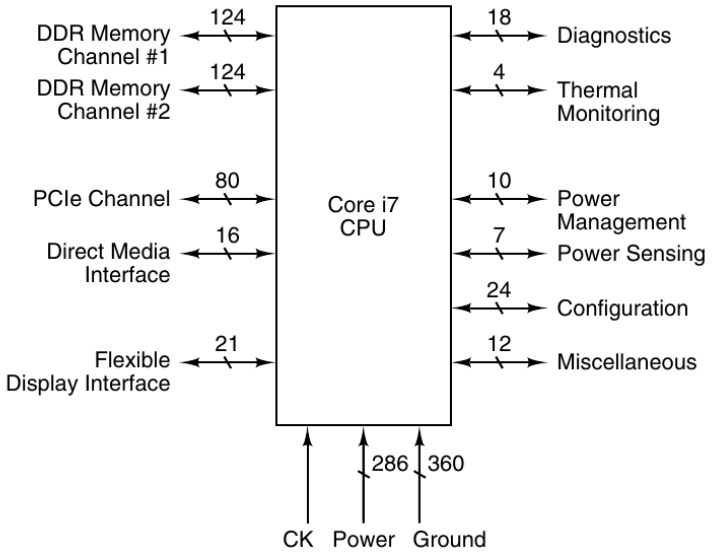


Figure 58: The logical pinout of an Intel CPU (i7).

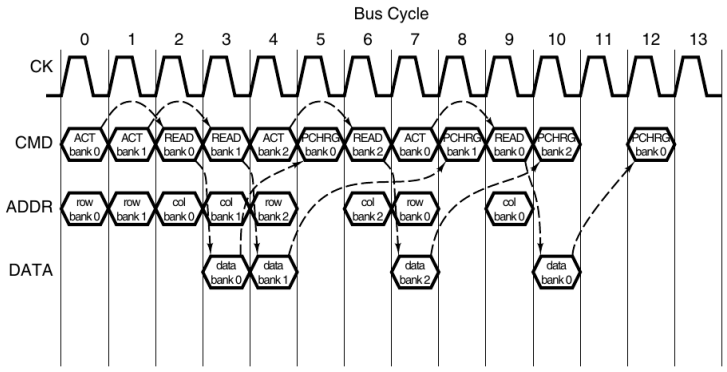


Figure 59: Pipelining DRAM memory requests on Core i7.

Intel i7 CPU: DRAM pipelining

- In general CPUs are faster than main memory (RAM),
- Pipelining the bus can help.
- Bus requests (called transactions) have six stages:
  1. The bus arbitration phase.
  2. The request phase.
  3. The error reporting phase.
  4. The snoop phase.
  5. The response phase.
  6. The data phase.
- Figure 59 illustrates pipelining the bus requests on an intel CPU.

UltraSPARC

- Although also used in workstations, the more important use of UltraSPARC processors is in large shared memory multiprocessor servers.
- Figure 60 illustrates an UltraSPARC III core.
- Later examples of the architecture are UltraSPARC T1, T2.
- Features:
  - 2 main internal L1 caches: 32 KB instructions, 64 KB data,
  - L2 caches off the chip (choose your own),

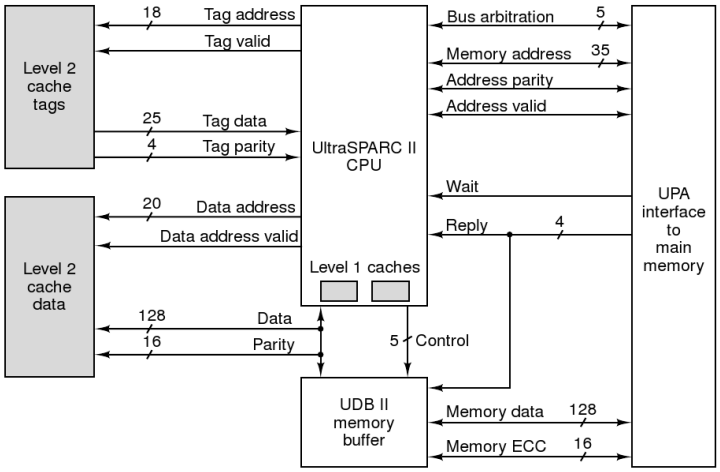


Figure 60: The main features of the core of an UltraSPARC system.

  - UPA (Ultra Port Architecture) connects (multiple) processors with (multiple) memories.
  - UltraSPARC Data Buffer II (UDB II) is a chip that is used to decouple the CPU from the memory, to allow them to work asynchronously.

picoJava

- picoJava / II was a research project at Sun Microsystem, never used commercially, but licensed to other companies (Fujitsu, Siemens, NEC).
- microJava II - a particular implementation (illustrated in Figure 61):
  - uses the PCI (convenience),
  - flash PROM interface (stores the program),
  - 64 bit wide memory bus, 32 bit wide PCI bus,
  - no level 2 cache.
  - runs native Java programs.
- A similar project is the ARM Jazelle extension.

Interfacing: Intel 8255A

- I/O chips connect I/O devices to the computer bus.

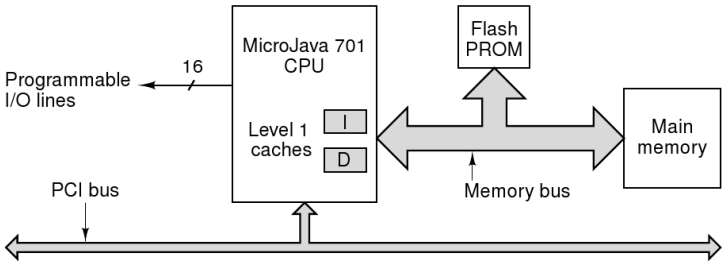


Figure 61: A microJava 701 system.

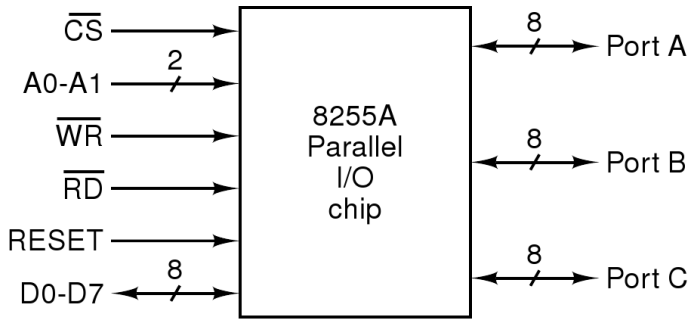


Figure 62: The Intel 8255a PIO chip.

- Example: UART (Universal Asynchronous Receiver Transmiter), PIO (Parallel Input/Output).
- Intel 8255A – typical PIO (illustrated in Figure 62):
  - can talk to registers on ports A, B, C.
  - or can be used for handshaking with external devices.
  - additional lines:
    - 8 bit data lines,
    - chip select,
    - address lines,
    - reset,
    - read, write lines.
  - to address more ports, several such chips may be cascaded.

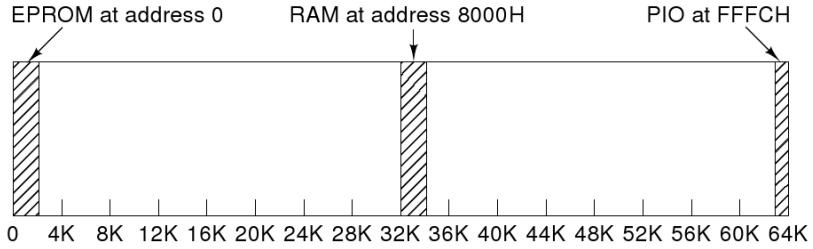


Figure 63: Location of EPROM, RAM and PIO in the 64KB address space of an enbedded device.

Address Decoding

- PIO chips can me configured as part of the I/O space or the memory space, as needed.
- memory-mapped I/O: assign 4 bytes of the memory space for the PIO (e.g. in a simple embedded device - toy), see Figure 63.
- the CS pin of the PIO is wired to the address lines of the bus (see Figure 64):
  - full address decoding - wiring to every line,
  - partial address decoding - wire only those lines that are relevant to the address space (but this ties up the address space - no memory upgrade).
- if the corresponding address is issued, the PIO takes it from the address lines of the bus.

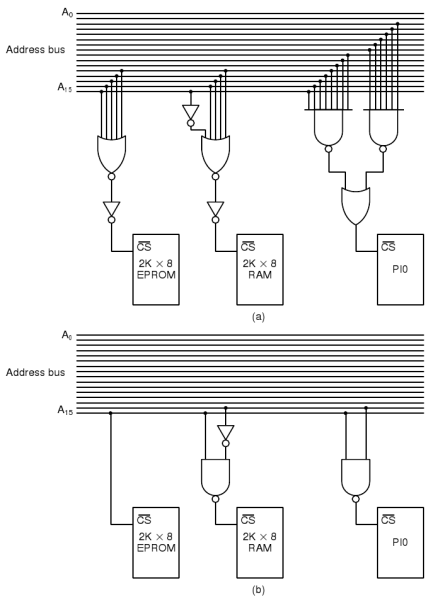


Figure 64: (a) Full address decoding. (b) Partial address decoding.

4 The Microarchitecture Level

4.1 An Example Microarchitecture

Overview - the microarchitecture level

- The job of the microarchitecture level is to implement ISA (the instruction set architecture), i.e. to provide a frame for the execution of the instructions that form ISA.
- The design and structure of the microarchitecture level depend on the ISA being implemented, as well as cost and performance goals of the computer
  - RISC designs: simple instructions that can be executed in one clock cycle,
  - CISC designs: complex instructions that take more than one clock cycle.

An example microarchitecture for IJVM

- Example microarchitecture: Mic-1, implementing IJVM (Integer Java Virtual Machine).
- JVM (Java Virtual Machine)/(Sun Microsystems) - a virtual machine that runs Java everywhere – Web pages, mobile phones, etc.
- The microarchitecture will contain:
  - a microprogram (in ROM), whose job is to fetch, decode and execute IJVM instructions.
  - state variables that can be accessed by all the functions of the microprogram (example: PC - the program counter).
- The instructions of IJVM are very simple, with just a few fields (usually one or two): opcode which identifies the type of instruction, and an operand field.

Data path

- The data path is the part of the CPU that contains the ALU, its inputs and outputs.
- Figure 65 illustrates the data path of our example microarchitecture:
  - it has 32 bit registers (with symbolic names assigned - PC, MDR, MAR, etc.) which are accessible only at the level of the microarchitecture,
  - the registers put their content on bus B (except the scratch register H, which has its own bus, A),

– the ALU performs logic-arithmetic operations, it has 6 controls (see Figure 66 for their combninations):
  * F_0, F_1 to determine the operation,
  * ENA, ENB enable operands,
  * INVA - inverting the left input,
  * INC - force a carry in the low order bit.

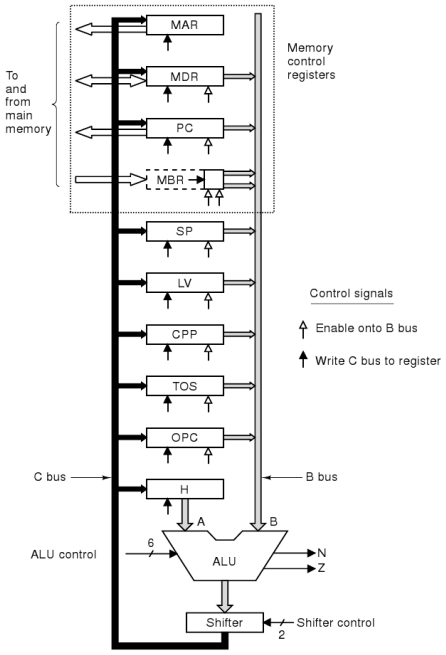


Figure 65: The data path of the example Mic-1 microarchitecture.

Data path timing

- The data flow on the data path (register-bus-ALU-register) determines implicit subcycles, see Figure 67:
  1. set up control signals - Δw,
  2. load register content on bus B - Δx,
  3. ALU and shifter operation - Δy,
  4. results propagate along bus C back to the registers - Δz.
- the ALU and shifter work all the time, but until $\Delta w + \Delta x + \Delta y$ they produce garbage.
- $\Delta w + \Delta x + \Delta y + \Delta z$ has to be smaller than the clock cycle.

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	1	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	–1

Figure 66: Useful control combinations for the ALU.

Memory operation

- There are two different ways to communicate with the memory:
  - a 32 bit word addressable port for ISA level data words.
  - a 8-bit byte addressable port for ISA level instructions.
- The 32 bit word addressable port:
  - controlled by 2 registers - MAR (Memory Address Register) and MDR (Memory Data Register),
  - MAR contains the address of the word that should be read into MDR.
- The 8 bit port is controlled by one register, PC (Program Counter), reading into the low-order 8 bit of the MBR (Memory Byte Register).
  - this can be either unsigned (and it is transformed into 32 bit values by adding 24 zeros in front), used for table indexing, or
  - signed (values from -128 to 127 trasformed into 32 bit words).

Data path control

- For controlling the data path the following (29) signals are needed, see Figure 65:

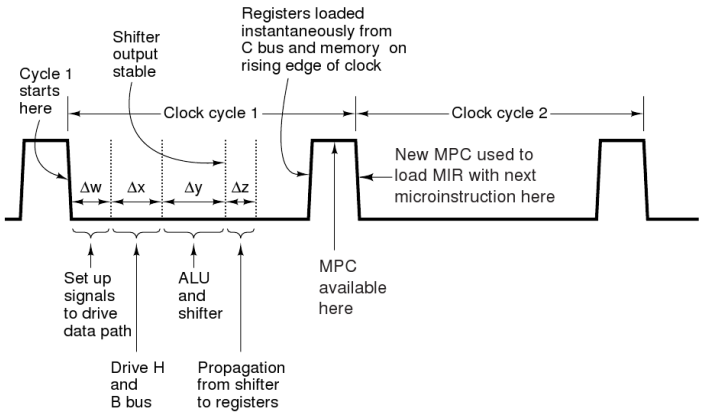


Figure 67: Timing diagram for a data path cycle.

  - 9 signals to control enabling registers onto the B bus for ALU input,
  - 9 signals to control writing data from the C bus,
  - 8 signals to control ALU and shifter,
  - 2 signals to indicate memory read/write via MAR/MDR (not shown),
  - 1 signal to indicate memory fetch via PC/MBR (not shown).

- The values of the signals specify one data path cycle: put values from the registers onto the B bus, carry out the ALU+shifter operations, store the results back into registers from C bus.

- Example operation - memory read:

  - assert memory read at the end of the cycle k (after loading MAR),
  - memory data is available at the very end of cycle $k + 1$,
  - memory data can be used at cycle $k + 2$ (from MDR),
  - and this is taking into account 100% cache hit rate.

Microinstructions

- A Mic-1 microinstruction packages together the signals needed to drive a data path cycle, and get the next microinstruction (for the next data path cycle).

- Figure 68 illustrates the structure of a Mic-1 microinstruction:

  - Addr contains the address of the potential next microinstruction,

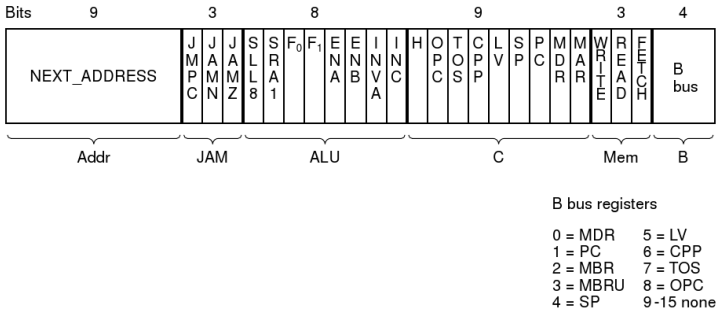


Figure 68: The format of a Mic-1 microinstruction.

- JAM - determines how the next microinstruction is selected, (JMPC - jump, JAMN - jump when negative, JAMZ - jump when zero)
- ALU - ALU and shifter controls,
- C - selects which registers are written from the C bus,
- Mem - controls the memory access,
- B - selects the source for the B bus (encodes one a value from 0 to 8 on 4 bits).

The Mic-1 microarchitecture

- Figure 69 illustrates the complete block diagram for the Mic-1 microarchitecture.
- Additional to the registers and signals already mentioned:
  - MPC (Microprogram Counter) - contains the address (in the control store) of the next microinstruction to be loaded,
  - MIR (Microinstruction Register) - contains the microinstruction driving the data path,
  - N, Z negative, zero (respectively) flags of the ALU.

Mic-1 operation

1 MIR is loaded from the word in the control store pointed by MPC during Δw, before Δx,

2 Control signals propagate from MIR into the data path:

- one register is loaded onto the B bus,

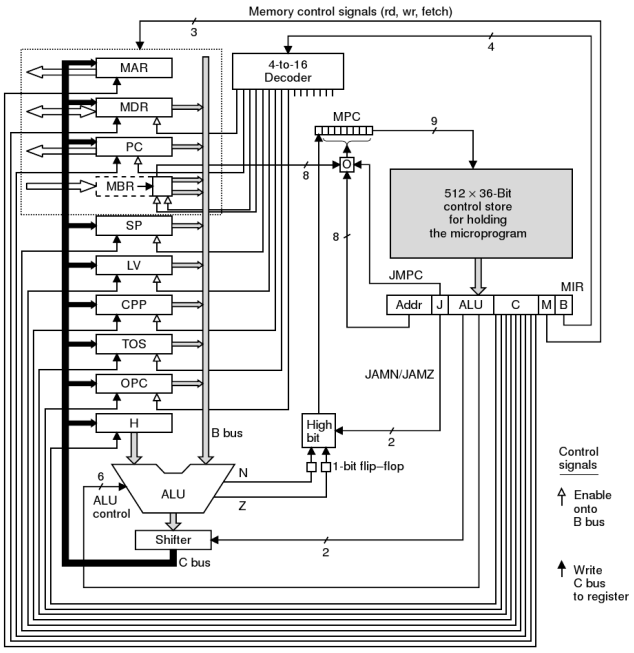


Figure 69: The Mic-1 microarchitecture.

- ALU is told which operation to perform,
- at $\Delta w + \Delta x$ the ALU inputs are stable.

3 ALU and shifter execute: at $\Delta w + \Delta x + \Delta y$ the ALU and shifter outputs are stable.

4 Write into registers (from C bus): by $\Delta w + \Delta x + \Delta y + \Delta z$ the results from ALU and shifter are in the registers, flipflops N, Z have been read and the next microinstruction is into MPC.

Microinstruction sequencing

- Microinstructions in the control store are not executed sequentially.
- Determining the next microinstruction:
  - copy the NEXT_ADDRESS to MPC and in the same time
  - inspect JAM:
    - Case JAM = 000 do nothing else, MPC points to the next microinstruction,
    - Case $JAM \neq 000$:
      MPC[8] = (JAMZ $\wedge$ Z) $\vee$ (JAMN $\wedge$ N) $\vee$ NEXT_ADDRESS[8])

– MPC can take two possible values (see Figure 70):
  * NEXT_ADDRESS,
  * NEXT_ADDRESS with the high-order bit or-ed with 1.
– When the JMPC bit is set, the 8 MBR bits are or-ed bitwise with the 8 lower-bits of NEXT_ADDRESS (box labelled "O" in Figure 69), which allows the implementation of multiway branch (jump) at any 256 addresses determined by the bits in MBR.

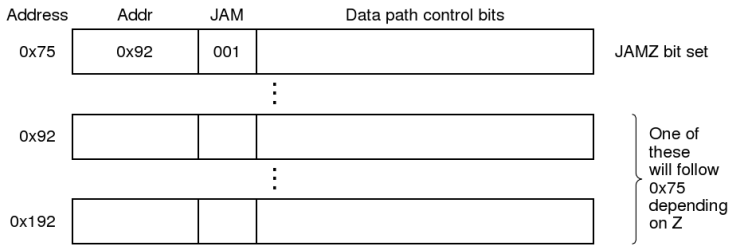


Figure 70: A microinstruction with JAMZ set to 1 has two potential successors.

4.2 An Example ISA: IJVM

IJVM

- IJVM - the ISA implemented by the Mic-1 (i.e. the macroarchitecture),
- stacks - memory area for variables:
  - local variables cannot be stored at absolute addresses,
  - LV - a register that points to the base of the local variable (frame),
  - SP - stack pointer,
  - LV - SP define a local variable frame,
  - variables are referred to by the offset from LV,
  - basic stack operations are PUSH, POP,
  - operand stacks are used for arithmetic operations (Figure 71 illustrates this use for $a1 = a2 + a3$).
  - each procedure has its own stack, see Figure 72.

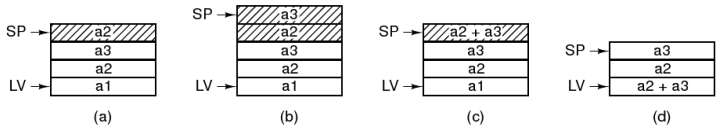


Figure 71: The use of an operand stack for doing arithmetic computation.

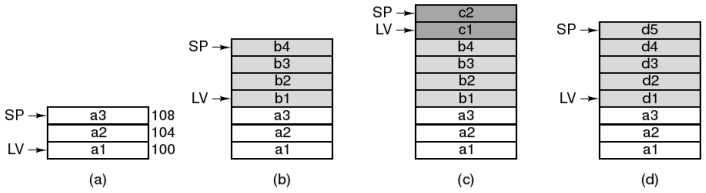


Figure 72: Use of a stack to store local variables. (a) While A is active. (b) After A calls B. (c) After B calls C. (d) After C and B return and A calls D.

The IJVM memory model

- The IJVM memory, illustrated in Figure 73 is partitioned in:
- The constant pool:

  - cannot be written by any IJVM program,
  - constants, strings, pointers to other memory areas live here,
  - CPP contains the pointer to this memory location.
- The local variable frame:
  - local variables of the program,
  - pointed to by the LV register.
- The operand stack pointed by SP (top address),
- The method area:
  - memory area containing the program,
  - referred to by PC.
- CPP, LV, SP address *words*, PC addresses *bytes*.

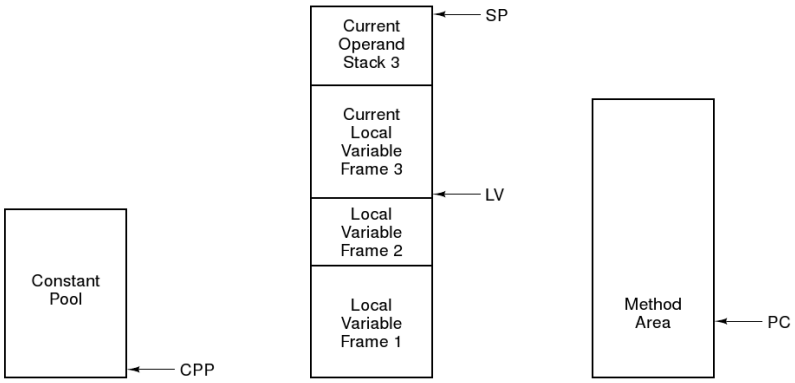


Figure 73: The IJVM memory model.

The IJVM instruction set

INVOKEVIRTUAL

- INVOKEVIRTUAL (see Figure 75) is an instruction for invoking another method:
  - push the pointer to the called object (procedure)
  - then push the parameters for the called object,
  - the parameter of INVOKEVIRTUAL (*disp*) contains:

Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Figure 74: The IJVM ISA instructions.

    - bytes 0-1: number of parameters,
    - bytes 2-3: size of local variable frame,
    - from byte 4, the first opcode (instruction) to be executed.
  - Executing INVOKEVIRTUAL amounts to:
    - setting SP to the top of the new stack,
    - store old LV and PC of the old frame,
    - set LV to the base of the new frame,
    - set PC to byte 4 (next instruction).

IRETURN

- Return from the procedure call:
  - deallocate space,
  - restore stack to former state,
  - the return value pushed on top of the stack,
  - restore the PC.
- The execution of IRETURN is represented in Figure 76.

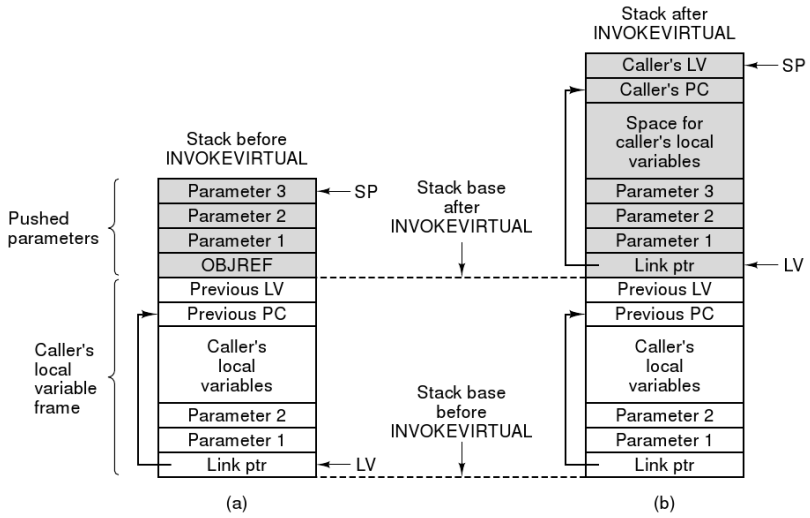


Figure 75: (a) Memory before executing INVOKEVIRTUAL. (b) Memory after.

Compiling JAVA to IJVM

- The example illustrated in Figure 77 shows how Java and IJVM relate.
- Java code is compiled to IJVM instructions.
- The IJVM instructions have a numeric format, assembly language representation is provided for easy reading by humans.
- Figure 78 illustrates the state of the memory when executing the example program.

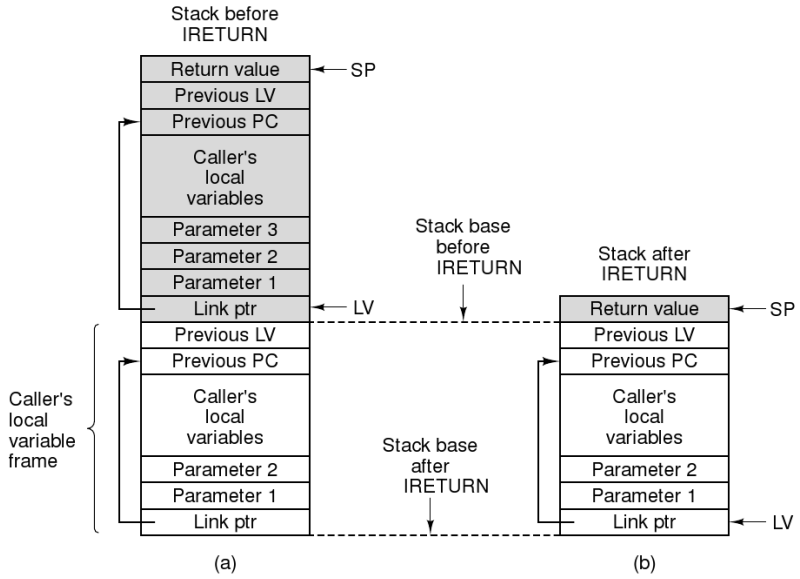


Figure 76: (a) Memory before executing IRETURN. (b) Memory after.

```
i = j + k;              1     ILOAD j    // i = j + k     0x15 0x02
if (i == 3)             2     ILOAD k                     0x15 0x03
    k = 0;              3     IADD                        0x60
else                    4     ISTORE i                    0x36 0x01
   j = j – 1;           5     ILOAD i    // if (i < 3)    0x15 0x01
                        6     BIPUSH 3                    0x10 0x03
                        7     IF_ICMPEQ L1                0x9F 0x00 0x0D
                        8     ILOAD j    // j = j – 1     0x15 0x02
                        9     BIPUSH 1                    0x10 0x01
                       10     ISUB                        0x64
                       11     ISTORE j                    0x36 0x02
                       12     GOTO L2                     0xA7 0x00 0x07
                       13  L1:           BIPUSH 0        // k = 0 0x10 0x00
                       14     ISTORE k                    0x36 0x03
                       15  L2:
        (a)                        (b)                          (c)
```

Figure 77: (a) Java fragment. (b) The corresponding Java assembly language. (c) The IJVM program in hexadecimal.

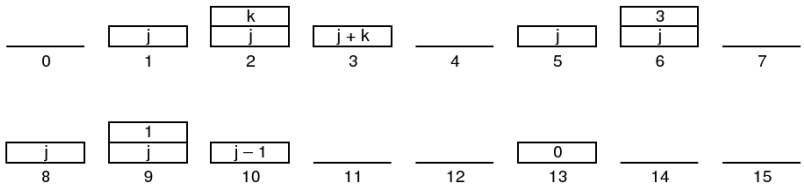


Figure 78: The stack after each instruction from Figure 77.

4.3 Implementation of the Instruction Set

Microinstructions: equational notation

- Our goal is to show how the instructions of the IJVM (Figure 74) are implemented at the level of the microarchitecture, by specifying the signals that go through the data path.
- To achieve the above, we specify an equational Micro Assembly Language (MAL), a high level language that reflects the characteristics of the architecture.
- Basic operations on the data path during one clock cycle (specified in MAL, illustrated in Figure 79):
  - one register is gated to the bus B,
  - one the A bus choices are -1, 0, 1 or the content of H,
  - one operation is selected for the ALU,
  - a register is written.
- Other operations allowed:
  - $\ll 8$ shifting bytes,
  - `goto` *label* for unconditional branching,
  - conditional branching (based on flags N, Z):

    ```
    if (Z) goto L1; else goto L2
    ```

  - conditional branching (based on JMPC): `goto (MBR or value)`.

Summary of Mic-1 registers

- The Mic-1 registers are used to describe the IJVM memory, or to hold operands:
  - MAR (Memory Address Register) - used to hold an address in the local variable frame. Used in connection with the memory access signals `rd` (the address from where to read), `wr` (the address where to write data).
  - MDR (Memory Data Register) - used to hold the data that is to be written into the memory (at address indicated by MAR), or where data is brought from memory (address indicated by MAR).
  - PC (Program Counter) - the address of the next instruction in the method area, used in connection with the `fetch` signals (which brings the opcode corresponding to the instruction into MBR),

DEST = H
DEST = SOURCE
DEST = $\overline{\text{H}}$
DEST = $\overline{\text{SOURCE}}$
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE – H
DEST = SOURCE – 1
DEST = –H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = –1

Figure 79: MAL description of permitted operations: SOURCE is a register that outputs on bus B, destination is a register that can be written from bus C.

- MBR (Memory Byte Register) - holds the opcode of the IJVM instruction pointed by PC.
- SP (Stack Pointer), the address of the top of the stack.
- LV (Local Variable), the address of the bottom of the local variable frame.
- CPP (Constant Pool Pointer), an address from the constant pool.
- TOS (Top of the Stack), the data contained in the memory location at the top of the stack (the one pointed by SP).
- OPC - scratch register, usually used to save the address of the opcode for a branch instruction while PC is incremented.

The Mic-1 microprogram

- The Mic-1 microprogram is illustrated in Figures 80,81,82,83,84.

Label	Operations	Comments
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch
nop1	goto Main1	Do nothing
iadd1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iadd2	H = TOS	H = top of stack
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Add top two words; write to top of stack
isub1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
isub2	H = TOS	H = top of stack
isub3	MDR = TOS = MDR – H; wr; goto Main1	Do subtraction; write to top of stack
iand1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iand2	H = TOS	H = top of stack
iand3	MDR = TOS = MDR AND H; wr; goto Main1	Do AND; write to new top of stack
ior1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
ior2	H = TOS	H = top of stack
ior3	MDR = TOS = MDR OR H; wr; goto Main1	Do OR; write to new top of stack
dup1	MAR = SP = SP + 1	Increment SP and copy to MAR
dup2	MDR = TOS; wr; goto Main1	Write new stack word
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
pop2		Wait for new TOS to be read from memory
pop3	TOS = MDR; goto Main1	Copy new word to TOS
swap1	MAR = SP – 1; rd	Set MAR to SP – 1; read 2nd word from stack
swap2	MAR = SP	Set MAR to top word
swap3	H = MDR; wr	Save TOS in H; write 2nd word to top of stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP – 1; wr	Set MAR to SP – 1; write as 2nd word on stack
swap6	TOS = H; goto Main1	Update TOS
bipush1	SP = MAR = SP + 1	MBR = the byte to push onto stack
bipush2	PC = PC + 1; fetch	Increment PC, fetch next opcode
bipush3	MDR = TOS = MBR; wr; goto Main1	Sign-extend constant and push on stack
iload1	H = LV	MBR contains index; copy LV to H
iload2	MAR = MBRU + H; rd	MAR = address of local variable to push
iload3	MAR = SP = SP + 1	SP points to new top of stack; prepare write
iload4	PC = PC + 1; fetch; wr	Inc PC; get next opcode; write top of stack
iload5	TOS = MDR; goto Main1	Update TOS
istore1	H = LV	MBR contains index; Copy LV to H
istore2	MAR = MBRU + H	MAR = address of local variable to store into
istore3	MDR = TOS; wr	Copy TOS to MDR; write word
istore4	SP = MAR = SP – 1; rd	Read in next-to-top word on stack
istore5	PC = PC + 1; fetch	Increment PC; fetch next opcode
istore6	TOS = MDR; goto Main1	Update TOS

Figure 80: The Mic-1 microprogram (1).

wide1	PC = PC + 1; fetch; goto (MBR OR 0x100)	Multiway branch with high bit set
wide_iload1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_iload2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_iload3	H = MBRU OR H	H = 16-bit index of local variable
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = address of local variable to push
wide_istore1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_istore2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_istore3	H = MBRU OR H	H = 16-bit index of local variable
wide_istore4	MAR = LV + H; goto istore3	MAR = address of local variable to store into
ldc_w1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
ldc_w2	H = MBRU << 8	H = 1st index byte << 8
ldc_w3	H = MBRU OR H	H = 16-bit index into constant pool
ldc_w4	MAR = H + CPP; rd; goto iload3	MAR = address of constant in pool

Figure 81: The Mic-1 microprogram (2).

Label	Operations	Comments
iinc1	H = LV	MBR contains index; Copy LV to H
iinc2	MAR = MBRU + H; rd	Copy LV + index to MAR; Read variable
iinc3	PC = PC + 1; fetch	Fetch constant
iinc4	H = MDR	Copy variable to H
iinc5	PC = PC + 1; fetch	Fetch next opcode
iinc6	MDR = MBR + H; wr; goto Main1	Put sum in MDR; update variable
goto1	OPC = PC – 1	Save address of opcode.
goto2	PC = PC + 1; fetch	MBR = 1st byte of offset; fetch 2nd byte
goto3	H = MBR << 8	Shift and save signed first byte in H
goto4	H = MBRU OR H	H = 16-bit branch offset
goto5	PC = OPC + H; fetch	Add offset to OPC
goto6	goto Main1	Wait for fetch of next opcode
iflt1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iflt2	OPC = TOS	Save TOS in OPC temporarily
iflt3	TOS = MDR	Put new top of stack in TOS
iflt4	N = OPC; if (N) goto T; else goto F	Branch on N bit
ifeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
ifeq2	OPC = TOS	Save TOS in OPC temporarily
ifeq3	TOS = MDR	Put new top of stack in TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Branch on Z bit
if_icmpeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
if_icmpeq2	MAR = SP = SP – 1	Set MAR to read in new top-of-stack
if_icmpeq3	H = MDR; rd	Copy second stack word to H
if_icmpeq4	OPC = TOS	Save TOS in OPC temporarily
if_icmpeq5	TOS = MDR	Put new top of stack in TOS
if_icmpeq6	Z = OPC – H; if (Z) goto T; else goto F	If top 2 words are equal, goto T, else goto F
T	OPC = PC – 1; fetch; goto goto2	Same as goto1; needed for target address
F	PC = PC + 1	Skip first offset byte
F2	PC = PC + 1; fetch	PC now points to next opcode
F3	goto Main1	Wait for fetch of opcode

Figure 82: The Mic-1 microprogram (3).

invokevirtual1	PC = PC + 1; fetch	MBR = index byte 1; inc. PC, get 2nd byte
invokevirtual2	H = MBRU << 8	Shift and save first byte in H
invokevirtual3	H = MBRU OR H	H = offset of method pointer from CPP
invokevirtual4	MAR = CPP + H; rd	Get pointer to method from CPP area
invokevirtual5	OPC = PC + 1	Save Return PC in OPC temporarily
invokevirtual6	PC = MDR; fetch	PC points to new method; get param count
invokevirtual7	PC = PC + 1; fetch	Fetch 2nd byte of parameter count
invokevirtual8	H = MBRU << 8	Shift and save first byte in H
invokevirtual9	H = MBRU OR H	H = number of parameters
invokevirtual10	PC = PC + 1; fetch	Fetch first byte of # locals
invokevirtual11	TOS = SP – H	TOS = address of OBJREF – 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = address of OBJREF (new LV)
invokevirtual13	PC = PC + 1; fetch	Fetch second byte of # locals
invokevirtual14	H = MBRU << 8	Shift and save first byte in H
invokevirtual15	H = MBRU OR H	H = # locals
invokevirtual16	MDR = SP + H + 1; wr	Overwrite OBJREF with link pointer
invokevirtual17	MAR = SP = MDR;	Set SP, MAR to location to hold old PC
invokevirtual18	MDR = OPC; wr	Save old PC above the local variables
invokevirtual19	MAR = SP = SP + 1	SP points to location to hold old LV
invokevirtual20	MDR = LV; wr	Save old LV above saved PC
invokevirtual21	PC = PC + 1; fetch	Fetch first opcode of new method.
invokevirtual22	LV = TOS; goto Main1	Set LV to point to LV Frame

Figure 83: The Mic-1 microprogram (4).

Label	Operations	Comments
ireturn1	MAR = SP = LV; rd	Reset SP, MAR to get link pointer
ireturn2		Wait for read
ireturn3	LV = MAR = MDR; rd	Set LV to link ptr; get old PC
ireturn4	MAR = LV + 1	Set MAR to read old LV
ireturn5	PC = MDR; rd; fetch	Restore PC; fetch next opcode
ireturn6	MAR = SP	Set MAR to write TOS
ireturn7	LV = MDR	Restore LV
ireturn8	MDR = TOS; wr; goto Main1	Save return value on original top of stack

Figure 84: The Mic-1 microprogram (5).

4.4 Designing the Microarchitecture Level

Design issues

- The design of the microarchitecture level is full of tradeoffs, of which the most important: speed vs. cost.
- Improving speed:
  - faster circuits (beyond the scope of this lecture),
  - increase the speed of execution by:
    - reducing the number of clock cycles needed to execute an (IJVM) instruction,
    - simplify the organization, such that the clock cycles can be shorter,
    - overlap the execution of instructions.
- Mic-1 is moderately simple and moderately fast.
- However, simple machines are not fast, and fast machines are not simple.

Merge the interpreter loop with the microcode

- Instead of executing the main fetch loop at the beginning of an instruction, do it at the end of each instruction, i.e. merge the interpreter loop into the end of each microcode sequence, see Figure 85.
- Moreover, in some cases, the interpreter loop may be used to take advantage of "dead" cycle (like it is the case for POP, see Figure 86).

Label	Operations	Comments
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
pop2		Wait for new TOS to be read from memory
pop3	TOS = MDR; goto Main1	Copy new word to TOS
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch

Figure 85: Main loop fused into the execution of POP.

Label	Operations	Comments
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
Main1.pop	PC = PC + 1; fetch	MBR holds opcode; fetch next byte
pop3	TOS = MDR; goto (MBR)	Copy new word to TOS; dispatch on opcode

Figure 86: Further integration of the interpreter loop.

A 3 bus architecture

- Allow bus A to be a full bus (i.e. allow all registers to access all buses).
- This way, 2 registers can be added.

Instruction Fetch Unit

- An instruction fetch unit (IFU):
  - is an independent unit for fetching and processing instructions:
    - increment the PC
    - fetch bytes from the byte stream,
    - assemble the operands.
  - it avoids the use of the ALU for fetching instructions and operands.
- There are two possible approaches:
  1. The IFU can interpret each opcode, determining how many additional fields must be fetched,
  2. The IFU makes available at all times the next 8 and 16 bit pieces whether or not it makes any sense to do so. The main execution unit can ask for what it needs.
- Figure 87 illustrates the data path of a microarchitecture with 3 buses and an instruction fetch unit (called Mic-2). The corresponding microprogram is shown in Figures 88,89,90.

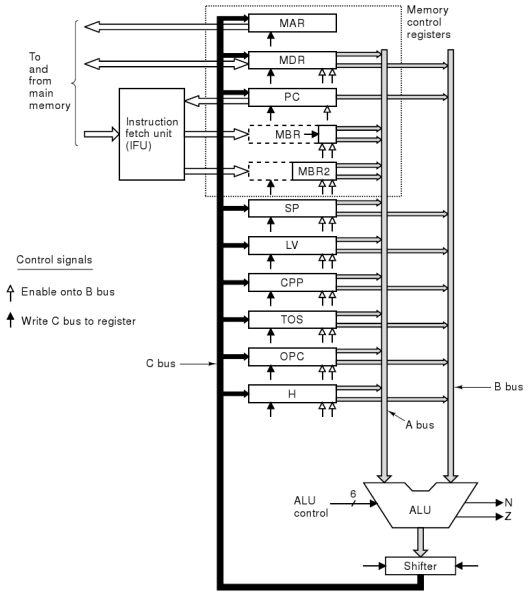


Figure 87: Mic-2, a 3 bus architecture with IFU.

Label	Operations	Comments
nop1	goto (MBR)	Branch to next instruction
iadd1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
iadd2	H = TOS	H = top of stack
iadd3	MDR = TOS = MDR+H; wr; goto (MBR1)	Add top two words; write to new top of stack
isub1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
isub2	H = TOS	H = top of stack
isub3	MDR = TOS = MDR−H; wr; goto (MBR1)	Subtract TOS from Fetched TOS-1
iand1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
iand2	H = TOS	H = top of stack
iand3	MDR = TOS = MDR AND H; wr; goto (MBR1)	AND Fetched TOS-1 with TOS
ior1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
ior2	H = TOS	H = top of stack
ior3	MDR = TOS = MDR OR H; wr; goto (MBR1)	OR Fetched TOS-1 with TOS
dup1	MAR = SP = SP + 1	Increment SP; copy to MAR
dup2	MDR = TOS; wr; goto (MBR1)	Write new stack word
pop1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
pop2		Wait for read
pop3	TOS = MDR; goto (MBR1)	Copy new word to TOS
swap1	MAR = SP − 1; rd	Read 2nd word from stack; set MAR to SP
swap2	MAR = SP	Prepare to write new 2nd word
swap3	H = MDR; wr	Save new TOS; write 2nd word to stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP − 1; wr	Write old TOS to 2nd place on stack
swap6	TOS = H; goto (MBR1)	Update TOS
bipush1	SP = MAR = SP + 1	Set up MAR for writing to new top of stack
bipush2	MDR = TOS = MBR1; wr; goto (MBR1)	Update stack in TOS and memory
iload1	MAR = LV + MBR1U; rd	Move LV + index to MAR; read operand
iload2	MAR = SP = SP + 1	Increment SP; Move new SP to MAR
iload3	TOS = MDR; wr; goto (MBR1)	Update stack in TOS and memory
istore1	MAR = LV + MBR1U	Set MAR to LV + index
istore2	MDR = TOS; wr	Copy TOS for storing
istore3	MAR = SP = SP − 1; rd	Decrement SP; read new TOS
istore4		Wait for read
istore5	TOS = MDR; goto (MBR1)	Update TOS
wide1	goto (MBR1 OR 0x100)	Next address is 0x100 Ored with opcode
wide_iload1	MAR = LV + MBR2U; rd; goto iload2	Identical to iload1 but using 2-byte index
wide_istore1	MAR = LV + MBR2U; goto istore2	Identical to istore1 but using 2-byte index
ldc_w1	MAR = CPP + MBR2U; rd; goto iload2	Same as wide_iload1 but indexing off CPP

Figure 88: Mic-2 microprogram (1).

iinc1	MAR = LV + MBR1U; rd	Set MAR to LV + index for read
iinc2	H = MBR1	Set H to constant
iinc3	MDR = MDR + H; wr; goto (MBR1)	Increment by constant and update
goto1	H = PC − 1	Copy PC to H
goto2	PC = H + MBR2	Add offset and update PC
goto3		Have to wait for IFU to fetch new opcode
goto4	goto (MBR1)	Dispatch to next instruction
iflt1	MAR = SP = SP − 1; rd	Read in next-to-top word on stack
iflt2	OPC = TOS	Save TOS in OPC temporarily
iflt3	TOS = MDR	Put new top of stack in TOS
iflt4	N = OPC; if (N) goto T; else goto F	Branch on N bit

Figure 89: Mic-2 microprogram (2).

Label	Operations	Comments
ifeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
ifeq2	OPC = TOS	Save TOS in OPC temporarily
ifeq3	TOS = MDR	Put new top of stack in TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Branch on Z bit
if_icmpeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
if_icmpeq2	MAR = SP = SP – 1	Set MAR to read in new top-of-stack
if_icmpeq3	H = MDR; rd	Copy second stack word to H
if_icmpeq4	OPC = TOS	Save TOS in OPC temporarily
if_icmpeq5	TOS = MDR	Put new top of stack in TOS
if_icmpeq6	Z = H – OPC; if (Z) goto T; else goto F	If top 2 words are equal, goto T, else goto F
T	H = PC – 1; goto goto2	Same as goto1
F	H = MBR2	Touch bytes in MBR2 to discard
F2	goto (MBR1)	
invokevirtual1	MAR = CPP + MBR2U; rd	Put address of method pointer in MAR
invokevirtual2	OPC = PC	Save Return PC in OPC
invokevirtual3	PC = MDR	Set PC to 1st byte of method code.
invokevirtual4	TOS = SP – MBR2U	TOS = address of OBJREF – 1
invokevirtual5	TOS = MAR = H = TOS + 1	TOS = address of OBJREF
invokevirtual6	MDR = SP + MBR2U + 1; wr	Overwrite OBJREF with link pointer
invokevirtual7	MAR = SP = MDR	Set SP, MAR to location to hold old PC
invokevirtual8	MDR = OPC; wr	Prepare to save old PC
invokevirtual9	MAR = SP = SP + 1	Inc. SP to point to location to hold old LV
invokevirtual10	MDR = LV; wr	Save old LV
invokevirtual11	LV = TOS; goto (MBR1)	Set LV to point to zeroth parameter.
ireturn1	MAR = SP = LV; rd	Reset SP, MAR to read Link ptr
ireturn2		Wait for link ptr
ireturn3	LV = MAR = MDR; rd	Set LV, MAR to link ptr; read old PC
ireturn4	MAR = LV + 1	Set MAR to point to old LV; read old LV
ireturn5	PC = MDR; rd	Restore PC
ireturn6	MAR = SP	
ireturn7	LV = MDR	Restore LV
ireturn8	MDR = TOS; wr; goto (MBR1)	Save return value on original top of stack

Figure 90: Mic-2 microprogram (3).

Pipelining

- Latches can be used to partition the data path (see Figure 91), so that multiple instructions can be executed in the same time (see Figure 92). This extension is Mic-3.
- Further refining the design, we have Mic-4, see Figure 93 with stages in the pipeline handling the sequencing of microoperations, see Figure 94.

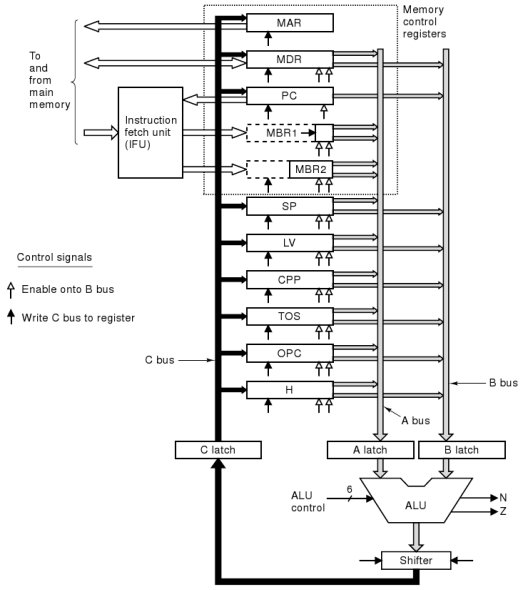


Figure 91: The 3 bus pipelined data path of Mic-3.

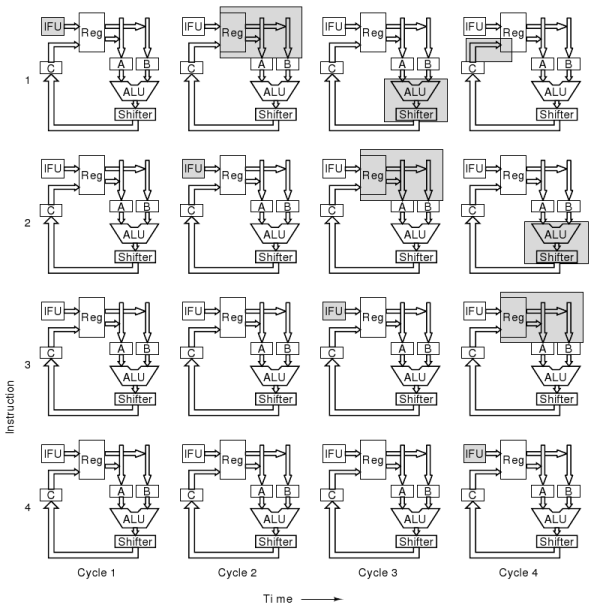


Figure 92: Graphical illustration of the Mic-3 pipeline.

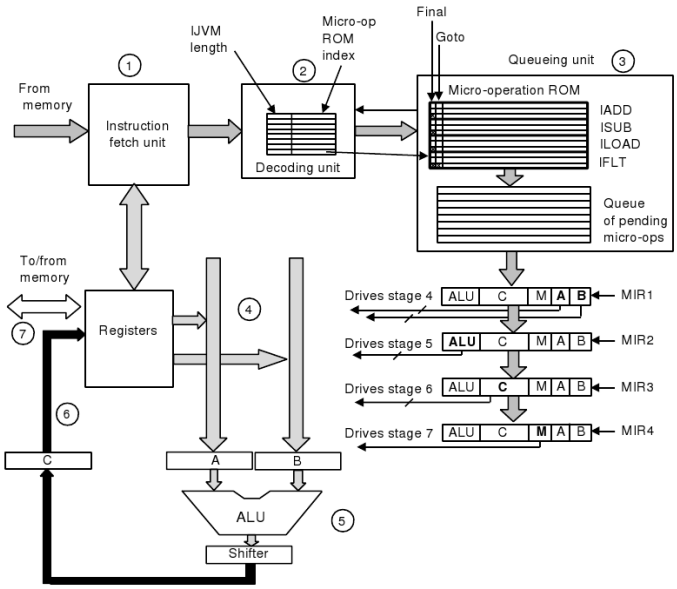


Figure 93: The main components of Mic-4.

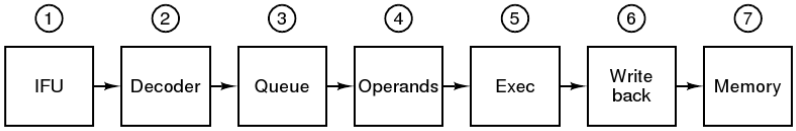


Figure 94: Mic-4 pipeline.

4.5 Improving Performance

Approaches to improving performance

- Cache memory.
- Branch prediction.
- Out-of-order execution and register renaming.
- Speculative execution.

Cache memory

- Cache: small, fast memory that holds the most recently used words.
- Improving bandwidth and latency: separate data and instrctions - split caches, especially level 1 caches.
- Locality principles:
  - spatial locality: memory locations close to the requested one are likely to be accessed and are brought from the memory.
  - temporal locality: recently addressed memory locations are accessed again, locations not recently accessed are discarded from the cache.
  - Taking into consideration the locality principles may improve programs significantly.
- Cache principles:
  - cache lines
  - when a memory word is referenced by a program, the cache controller checks whether the word is in the cache, if not it brings it with its own cache line, after making room by discarding some cache lines.

Direct-mapped caches

- Simplest form of cache, see Figure 95.
- Each cache entry consists of:
  - a valid bit indicating whether the entry has valid data,
  - a tag field consisting of a unique value identifying the memory line,
  - a data field holding a cache line of 32 bytes.
- A word in memory can be stored in a single position in cache:
  - the LINE field identifies in which cache line it is,
  - the TAG field checks whether the right line is in the cache,

– the WORD field determines the position of the word in the cache line.

- Problem with direct mapped caches:
  – if two words in the memory, which use the same position in the cache (or even the same cache line) both are heavily used (i.e. need to be in the cache),
  – then the respective line will have to be discarded and loaded very often (and the advantages of caching are lost).

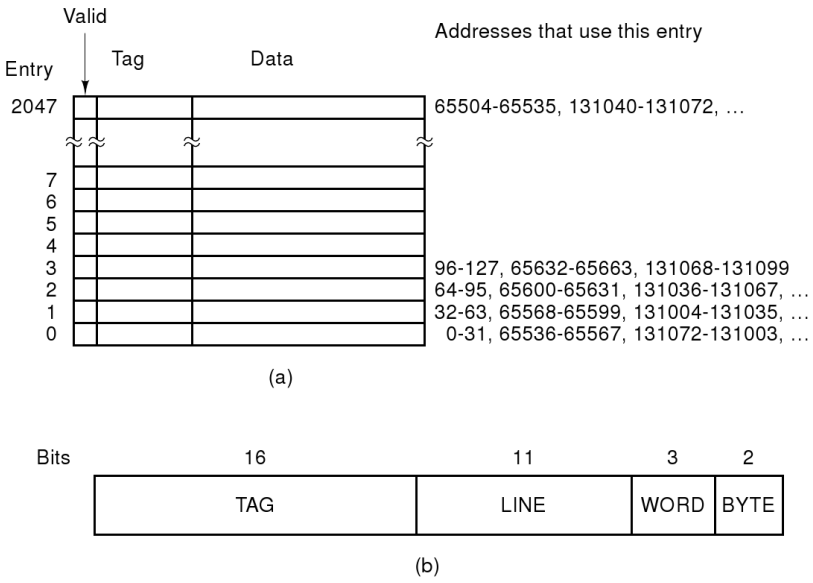


Figure 95: (a) A direct mapped cache. (b) A 32 bit virtual address mapping memory words into the cache.

Set-associative caches

- Allow one word in the memory to be stored in various (fixed number n of) places in the cache, see Figure 96.
- Usually $n = 2$ or $n = 4$.
- When a word has to be brought into the memory, but the corresponding cache line is occupied, the other alternatives are tried.
- If all the lines are occupied, the least recently used (LRU) line is discarded from the cache to make room for the line coming from the memory.

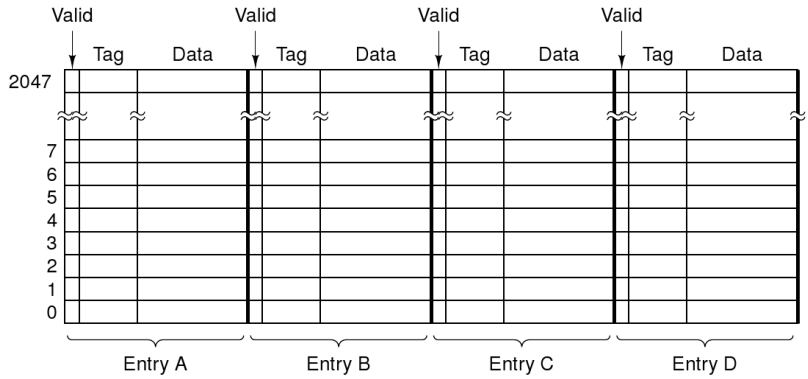


Figure 96: A four way associative cache.

Branch prediction

- Real world code is full of branch instructions, see Figure 97, also remember pipelines, like the one for Mic-4, Figure 94:
  - unconditional branches (BR) create problems:
    - instruction decoding occurs in the second stage in the pipeline, while the first stage already fetches the next instruction, although a jump is performed
    - the next instruction can be put into a delay slot - it is still executed but the result will be used if/when it is needed.
  - for conditional branches:
    - (BNE) the fetch unit does not know which instruction to load until much later in the pipeline,
    - the pipeline has to stall until it is known whether the branch is taken or not.
  - The problems are addressed by using branch prediction techniques.

- Dynamic branch prediction: the CPU guesses whether a branch is taken using a cache-like history table.
- Possible organizations of the history tables are illustrated in Figure 98:
  - using 1 bit predictor, saying whether the branch was taken last time or not,
  - using 2 bits for prediction, the second bit says whether the guess was true last time - if the second bit is 0, the first bit is changed, so the first error (end of loop) does not change prediction,

```
if (i == 0)                    CMP i,0; compare i to 0
   k = 1;                      BNE Else; branch to Else if not equal
else                           Then:   MOV k,1; move 1 to k
   k = 2;                      BR Next; unconditional branch to Next
                          Else: MOV k,2; move 2 to k
                          Next:
   (a)                              (b)
```

Figure 97: (a) A program fragment. (b) Corresponding translation into generic assembly language, with branching instructions.

  - prediction bits plus target address of the last branch.

- Static branch prediction requires previous knowledge about the structure of the program, i.e. it requires special compiler support.

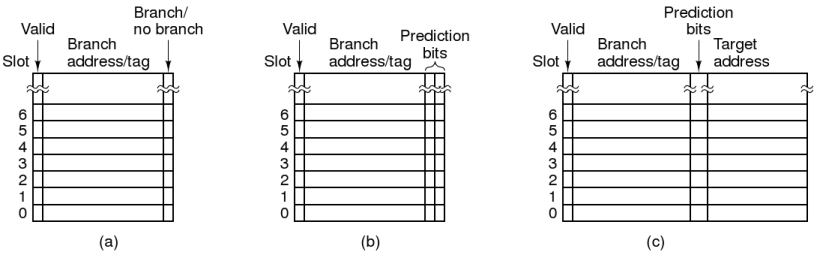


Figure 98: (a) A 1-bit branch history. (b) A 2-bit branch history. (c) Mapping betwern branch history address and target address.

Out-of-order execution

- Modern CPUs a re superscalar.
- The decoding unit feeds multiple functional units.
- In-order execution:
  - instructions are issued in the order they appear in the program, e.g. 1-2-3-4.
  - instruction 2 may depend on instruction 1 and has to wait for it to finish execution,
  - however, if instruction 3 does not depend of the previous ones, but it will wait as well

- Out-of-order execution:
  - the decode unit may change the order of the instructions (e.g. 1-3-2-4).
  - problem: the same register R may be written both by instruction 1 and 3.
- Register renaming: the CPU has secret registers, invisible to the programs:
  - the decode unit may change the instructions to read/write from/to secret registers,
  - instruction 3 is changed to write to S,
  - instruction 4 is changed from reading from R to reading from S.

Speculative execution

- Out-of-order execution only operates within basic blocks.
- Speculative execution allows the code to be moved beyond the block boundaries.
- Figure 99 illustrates a code sequence and its partition into code blocks.
- With speculative execution:
  - code may be executed before it is known that it will be needed,
  - both branches of the conditional may be executed simultaneously with the block that computes the condition.
- Speculative execution requires additional instruction and compiler support.

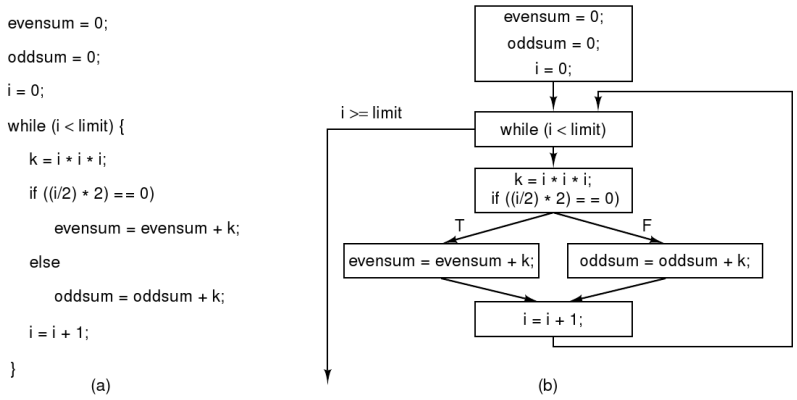


Figure 99: (a) Program fragment . (b) Corresponding basic block graph.

4.6 Example Microarchitectures

Intel i7

- Figures 100, 101, illustrate the Intel Pentium 2 microarchitecture, with details of the instruction fetch unit (IFU) and the dispatch unit.
- Although this is an ancient processor, it shows how the design incorporates the techiques mentioned above.
- The structure of the instruction set (Intelx86 has CISC instructions) influences the complexity of the various components of the microarchitecture.

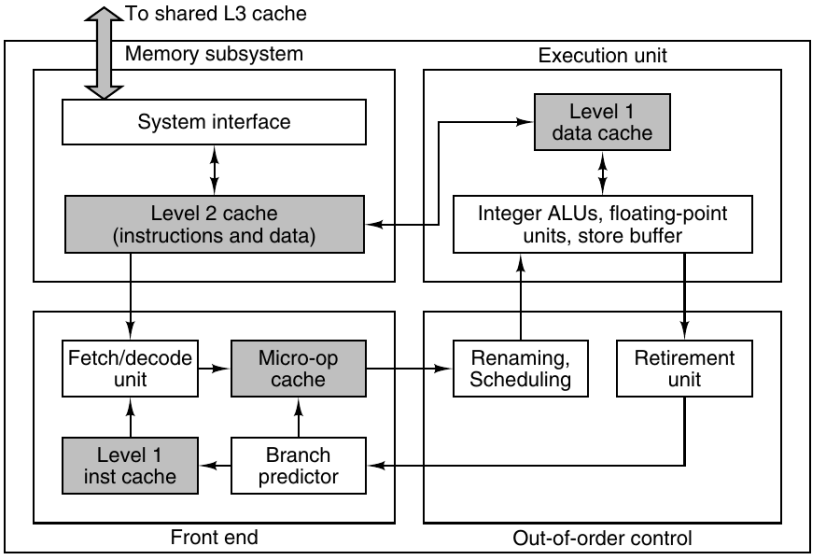


Figure 100: Block diagram of the i7 microarchitecture.

OMAP4430 (ARM)

- Figures 102, 103 illustrate the microarchitecture and pipeline of OMAP4430 (ARM), a RISC machine.

ATmega168 microcontroller

- Figure 104 illustrate the microarchitecture of ATmega168 microcontroller.

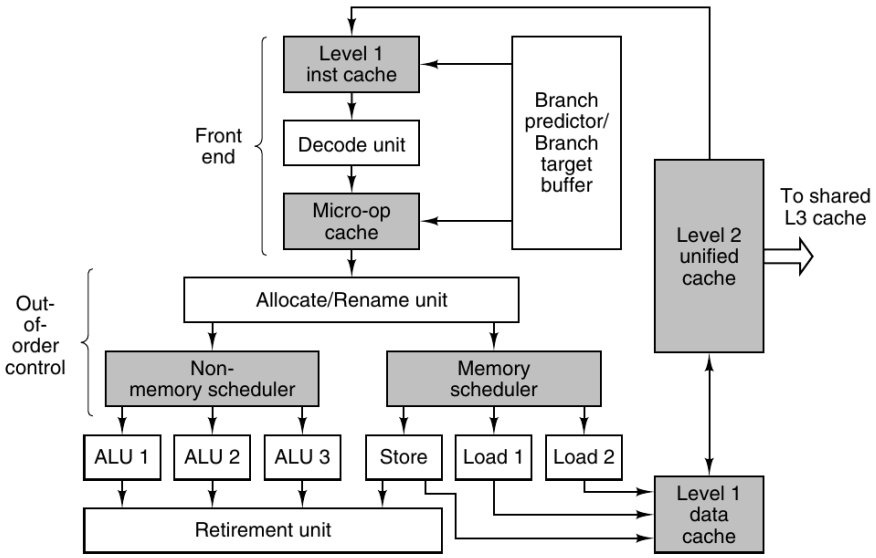


Figure 101: Simplified pipeline of the i7 microarchitecture.

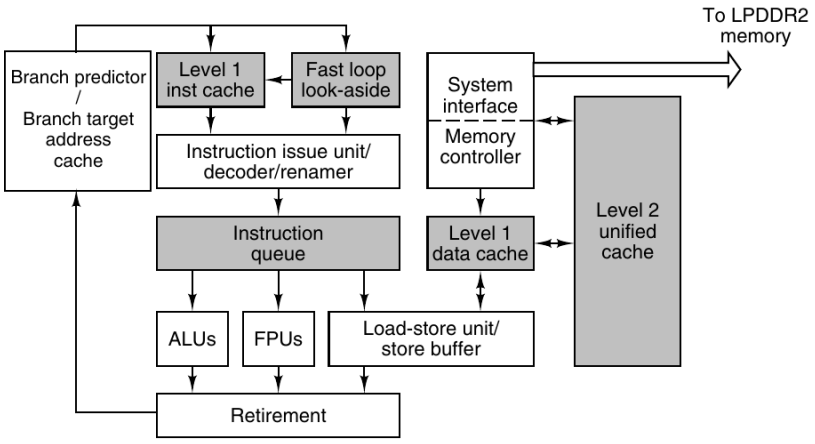


Figure 102: The OMAP4430 microarchitecture.

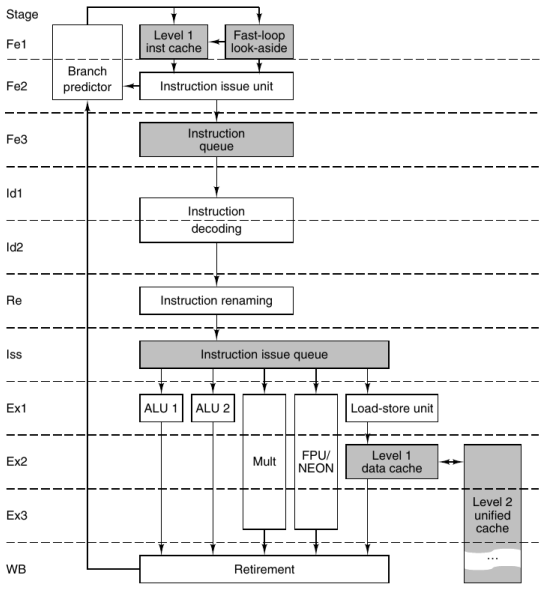


Figure 103: The pipeline of OMAP4430.

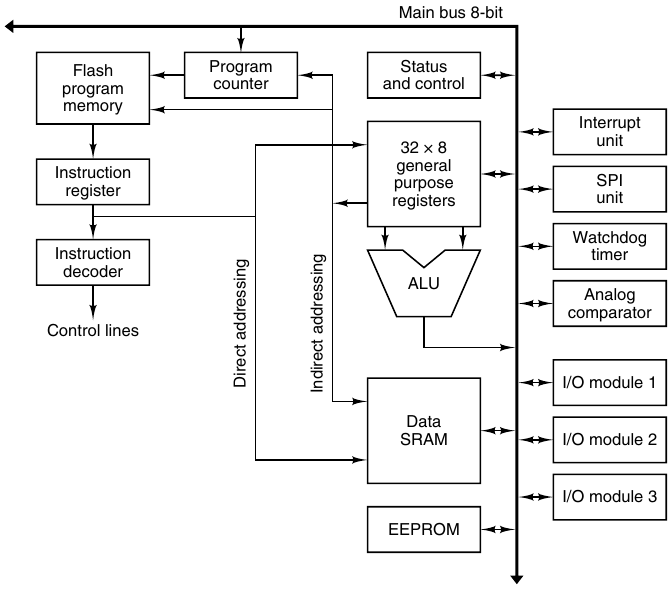


Figure 104: The microarchitecture of ATmega168 microcontroller.

5 The Instruction Set Architecture Level

5.1 Overview of the Instruction Set Architecture Level

The Instruction Set Architecture

- The Instruction Set Architecture level (ISA) = essentially, the set of instructions that can be executed on a machine.
- Initially ISA was the only level of the machine.
- It is referred to as "the architecture".
- While it is possible to build machines that execute high level language programs (C++, etc.), this may lead to loss of performance (compilation vs. interpretation), and such machines would only run programs written in the respective language.
- The common approach: programs written in the high level languages are translated into a common intermediate form - the ISA level, and hardware executes the ISA-level instructions.
- A representation of the ISA level is given in Figure 105.

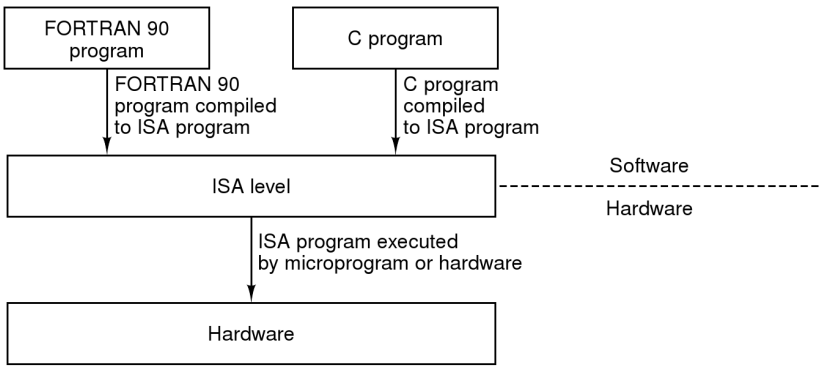


Figure 105: The ISA level - the interface between high level programming language and hardware.

ISA design

- In theory, designing the ISA level should be a compromise between the needs that come from the high level languages (efficient "hardware" execution of some instructions) and the possibilities of the implementation in hardware.

- In practice, backward compatibility (i.e. old code should run on the new machine, old instructions should be present) is a major factor in the design of the ISA.
- ISA design factors:
  - a good ISA should define a set of instructions that can be implemented efficiently in current and future technologies,
  - a good ISA should provide a clean target for compiled code (regularity and completeness).

Properties of the ISA level

- Tanenbaum: "since the ISA is the interface between the hardware and the software, it should make hardware designers happy (easy to implement) and make the software designers happy (easy to generate good code for)."
- The ISA level is defined by how the machine appears to a machine language programmer (but not many people do that anymore).
- The ISA-level code can be redefined as the compiler's output.
- To output ISA code, a compiler has to know
  - the memory model,
  - registers,
  - data types,
  - instructions

  of the architecture.
- In principle other details are not relevant (e.g. pipelined, microprogrammed, superscalar, etc.).
- However, in practice some of the details are important (e.g. the compiler can issue alternating integer and floating point instructions for a superscalar design where these can be handled in parallel).

Architecture documents

- For some (most) architectures, the ISA level is specified in a document provided by:
  - the manufacturer (Intel),
  - an industry consortium (V9 SPARC, JVM).
- These documents contain normative sections (imposing requirements) and informative sections.

- For most machines there are at least 2 modes:
  - kernel mode, to run the operating system and all instructions,
  - user mode, that restricts the instructions that can be executed.

5.2 Memory models

- Memory is divided into memory cells, each with its own address.
- Most widely used are 8 bit cells (bytes), but cells from 1 to 60 bits have been used.
- Bytes are grouped into 4-byte or 8-byte words, with instructions manipulating entire words.
- Many architectures require words to be aligned to natural boundaries.
- Alignment is required because memories operate more efficiently.
- Non-alignment is often present due to backward compatibility (e.g. Pentium 4 needing to run 8080 code).
- Figure 106 illustrates memory alignment.
- Most machines have a linear address space, some have separate address space for data and for instructions.
- Memory semantics: LOAD after STORE - serialized memory requests, SYNC instructions (to block any other memory access until the current one has ended).

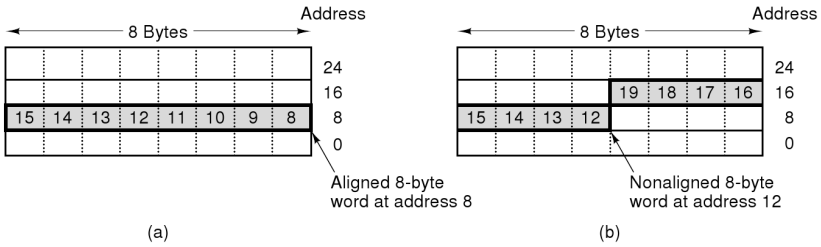


Figure 106: (a) Aligned memory. (b) Non-aligned memory.

5.3 Registers

- All computers have registers visible at the ISA level (however not all registers visible at the microarchitecture level will be visible from ISA),
- Two types of registers:
  - special purpose: stack pointer, program counter, etc.
  - general purpose: hold local variables, intermediate results, etc.
- On some machines, these types of registers are interchangeable (e.g. on SPARC R1-R25).
- Even when registers are interchangeable, there may be some conventions on how to use them (indicated in the ISA specification document).
- Some registers are only available in kernel mode (registers that control the cache, I/O functionality, other hardware functions).
- PSW (Program Status Word) or flags registers - hybrid kernel/user register containing various miscellaneous bits needed by the CPU:
  - N - the result was Negative,
  - Z - the result was Zero,
  - V - the result was an oVerflow,
  - C - the result caused a Carry in the leftmost bit,
  - P - the result had even Parity.

Memory, registers - example: IA32

- IA32(x86) can be traced back to 8088 (8008, 4004).
- It has 3 operating modes:
  1. real mode: same as 8088 (if any program does anything wrong, it crashes),
  2. virtual 8086 mode: run 8088 programs in a protected mode, the OS creates and environment where these programs are run, eliminating system crashes.
  3. protected mode: full 32 bit instruction set, 4 privilege levels (0 - operating system, 3 - user, 1,2 hardly ever used).
- Address space: 16,384 segments, each from address 0 to $2^{32} - 1$, only segment 0 used by most OS (Windows, Unix), 32 bit words.
- Figure 107 illustrates the IA 32 registers:
  - general purpose, 32 bit:

    - EAX (arithmetic),
    - EBX (pointers),
    - ECX (looping),
    - EDX (multiplication and division, together with EAX)

  each also holding 8 and 16 bit registers,
- general (but more restricted) purpose, 32 bit registers:
    - ESI, EDI (hold pointers into memory),
    - EBP (stack base pointer),
    - ESP (stack pointer),
- segment registers (CS to GS), for backward compatibility,
- EIP - the program counter, EFLAGS - the PSW.

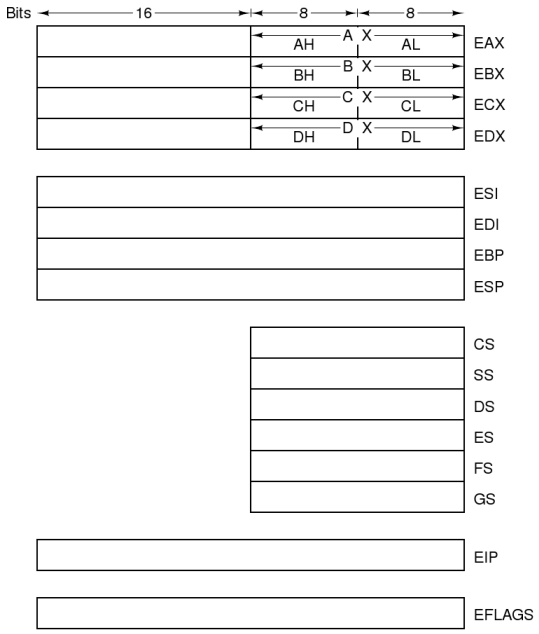


Figure 107: The registers of the IA 32 architecture.

Memory, registers - example: UltraSPARC II

- UltraSPARC II is a 64 bit machine compliant with SPARC 9 specifications.
- 2^{64} addressable array of memory.
- The SPARC ISA is clean, though the organization of registers is somewhat complex, to make procedure calls more efficient.

- Registers: 32 64-bit general purpose registers (R0-R31), see Figure 108, 32 floating point registers.
- The use of registers indicated in Figure 108 is strongly recommended "unles you have a Black Belt in SPARC Guru and really, really know what you are doing."
- Floating point registers can be used for single precision (32 bit), double precision (64 bit), or grouped together for quad precision (128 bits).

Register	Alt. name	Function
R0	G0	Hardwired to 0. Stores into it are just ignored.
R1 – R7	G1 – G7	Holds global variables
R8 – R13	O0 – O5	Holds parameters to the procedure being called
R14	SP	Stack pointer
R15	O7	Scratch register
R16 – R23	L0 – L7	Holds local variables for the current procedure
R24 – R29	I0 – I5	Holds incoming parameters
R30	FP	Pointer to the base of the current stack frame
R31	I7	Holds return address for the current procedure

Figure 108: Registers of UltraSPARC II.

- Implementation of procedure calls (see Figure 109):
  - 32 general purpose registers are always visible,
  - register windows implement efficient procedure calls,
  - the number of the register window is stored in the CWP,
  - this technique requires register renaming.
- UltraSPARC II is a load/store architecture, i.e. only LOAD and STORE access memory directly.

Memory, registers - JVM

- Memory model: 4 main regions:
  - local variable frame
  - the operand stack,
  - the method area,
  - the constant pool

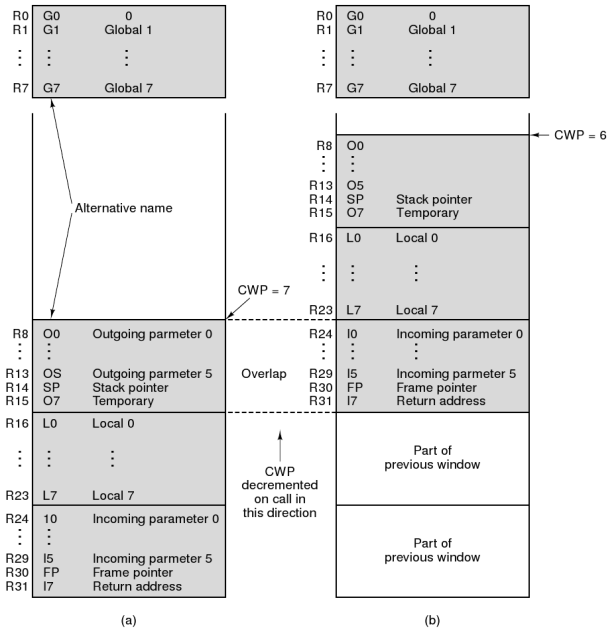


Figure 109: UltraSPARC II register window.

(pointed respectively by LV, SP, PC, CPP).

- All memory access is made by offsets of the above registers (no pointers or absolute memory is used – Java code should run everywhere, but not spy).
- Each memory region is limited (64 Kb), so for dynamically created data structures use the heap (e.g. `int a[] = new int[4096]`).
- The garbage collector cleans up the heap when its allocated space runs out.
- JVM does not have any general-purpose registers that can be loaded or stored under program control, it is a pure stack machine.
- The problem with stack machines is that they require a large number of memory references.
- To deal with the above problem, multiple instructions are folded together.

5.4 Data Types

- All computers process data, the main issue is whether a data type is hardware supported.

- Numeric data types:
  - integer: 8, 16, 32, 64 bits (most represented in two's complement), signed or unsigned,
  - floating point: 32, 64, 128 bits,
  - decimal (Cobol friendly machines).
- Non-numeric data types:
  - characters (ASCII, UNICODE),
  - boolean,
  - pointers.

Data types examples

- Intel: integers, floating point (see Figure 110), characters, strings.
- UltraSPARC II: integers, floating point (see Figure 111), no support for characters.
- JVM: Java is a strongly typed language, every object has a type (numerical types in Figure 112).

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×		
Unsigned integer	×	×	×		
Binary coded decimal integer	×				
Floating point			×	×	

Figure 110: Numerical data types for the Intel architecture.

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×	×	
Unsigned integer	×	×	×	×	
Binary coded decimal integer					
Floating point			×	×	×

Figure 111: Numerical data types of UltraSPARC II.

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×	×	
Unsigned integer					
Binary coded decimal integer					
Floating point			×	×	

Figure 112: Numerical data types of JVM.

5.5 Instruction Formats

- An instruction consists of an opcode, possibly together with additional information, such as where the operands come from, where they go (i.e. addressing).
- Figure 113 illustrates the possible formats of an instruction.
- Instructions can take one, a subdivision, or several memory words, see Figure 114.

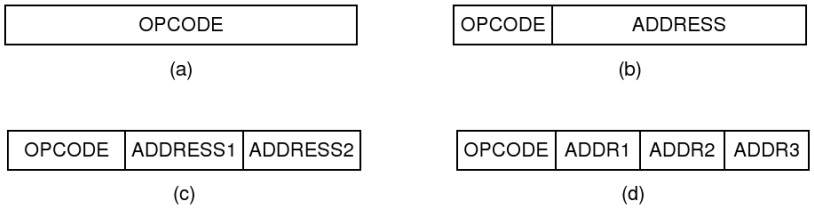


Figure 113: Common instruction formats: (a) Zero-address instruction. (b) One-address instruction. (c) Two address instruction. (d) Three-address instruction.

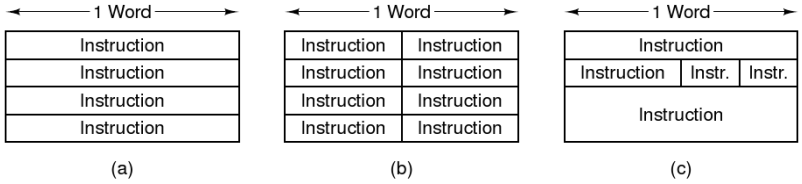


Figure 114: Possible relationships between instructions and word length.

Design criteria for instruction formats

- Instruction design depends on technology:

  - fast memory (or desire to produce cheap processors) suggests a stack based design: JVM
  - slow memory suggests a design with many registers: SPARC.
- Short instructions are better than longer instructions (with respect to memory space).
- However, short instructions may be difficult to decode.
- Memory bandwidth bottlenecks favours short instructions.
- The opcode should allow the extension of the instruction set.
- Number of bits in an address field (bytes vs. words) - short addresses and good memory resolution can be traded against eachother.

Expanding opcodes

- Figure 115 show how to choose opcodes for 16 bit instructions.
- In practice, choosing opcodes is not as regular as this.
- Key approaches:
  - shorter opcodes for the most widely used instructions (minimize the average length),
  - shorter opcodes for instructions that need longer addresses.

Example instruction format: Intel IA32

- The Intel 32 bit instructions are highly irregular, with up to six variable length fields, five of which are optional, see Figure 116.
- The prefix allowed the extension of the opcode when Intel ran out of codes for instructions.
- Mode contains 3 regions, first (rightmost) being sometimes used as an extension of the opcode (11 bit), mode allowing only 4 ways to address operands, one of them always being a register, etc.
- This design is the result of backward compatibility requirements.

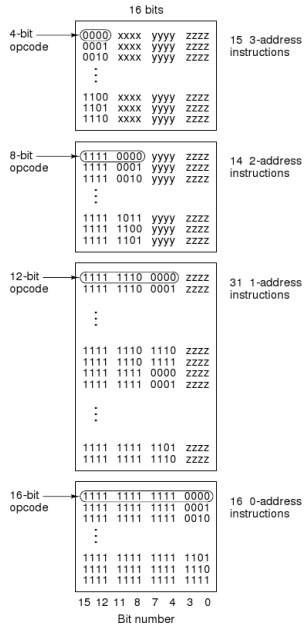


Figure 115: An expanding opcode with 15 3 address instructions, 14 2 address instructions, 31 one address instructions and 16 zero address instructions. The fields xxxx, yyyy, zzzz are 4 bit address fields.

Example instruction formats: UltraSPARC

- Simple instructions, aligned to memory, see Figure 117
- In time, more instructions were added, however, most conform to the format.
- First two bits are used to determine the instruction format and tell hardware where to get the rest of the opcode:

  1a both sources are registers,

  1b 1 register and one constant (± 4096), destination is a register,

  2 places in a register a constant,

  3 nonpredictive loops,

  4 the CALL instruction.

Example instruction formats: JVM

- Very simple instructions, see Figure 118.

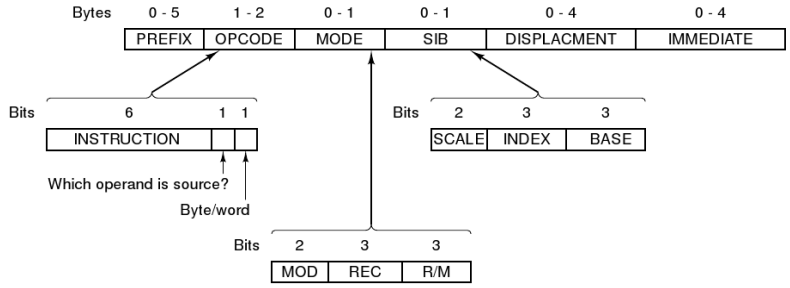


Figure 116: Format of the Intel 32 bit instruction.

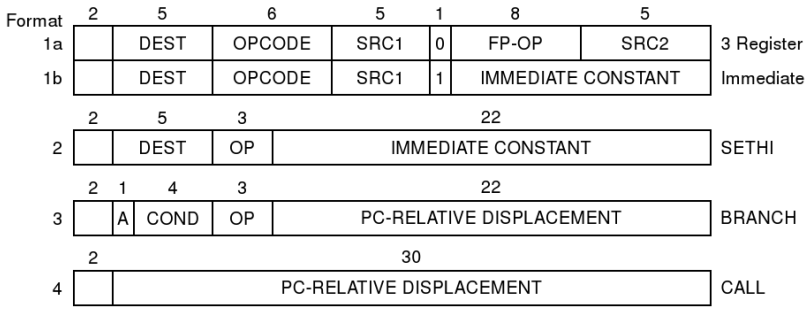


Figure 117: The instruction format of UltraSPARC.

- Formats 1, 2, 3 are used for most instructions (all but 8 instructions).
- Special cases of instructions of type 2, 3 were coded as format 1: `ILOAD_0 [1, 2, 3, 4, 5]` for `ILOAD 0 [1, 2, 3, 4, 5]`.

5.6 Addressing

- Addressing modes are about the sources of operands in instructions.
- Example: possible implementations of `ADD DESTINATION, SOURCE1, SOURCE2`:
  - using memory locations (expensive, memory addresses are large),
  - using registers (expensive, values have to be loaded into registers)

    `DESTINATION = SOURCE1 + SOURCE2`
    or better, with a destructive variant:
    `REGISTER1 = REGISTER2 + SOURCE1`
    even better, use the accumulator to store one operand,
    and even better, use the stack (no registers needed).

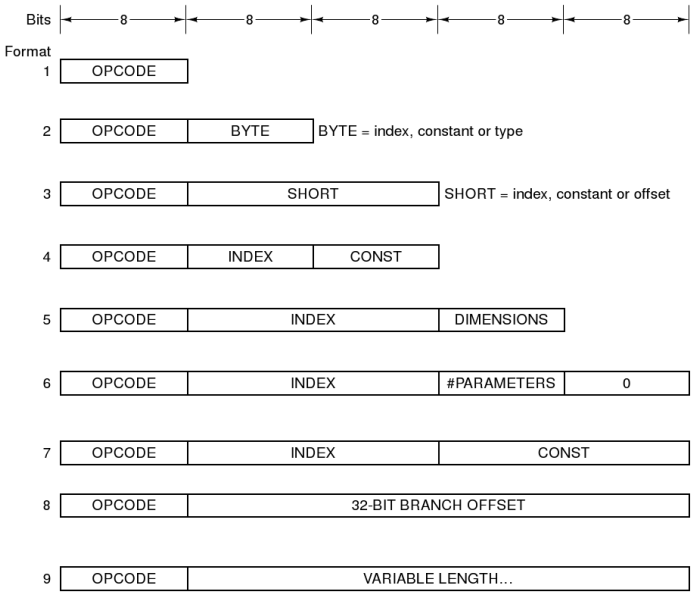


Figure 118: JVM instructions format.

Addressing Modes

- Immediate addressing – specify an operand as part of the instruction (used to specify a small integer constant):

```
MOV R1 4.
```

- Direct addressing – give the full address of the operand in the memory (used to access global variables with known values at compile time).
- Register addressing – conceptually the same as direct addressing, but use registers instead of a memory location (nearly all instructions of UltraSPARC II use this mode).
- Register indirect addressing - the operand comes or goes from/to memory, but the address is contained in a register (pointers), as illustrated in Figure 119.
- Indexed addressing - refer to memory words as an offset of a register (see Figure 120 for an example, R1 is used to hold the accumulated OR, R2 is the index to step through the array, R3 holds the constant 4096 (the lowest index not to be used, R4 is a scratch register, to hold each result as it is formed)).

```
        MOV R1,#0     ; accumulate the sum in R1, initially 0
        MOV R2,#A     ; R2 = address of the array A
        MOV R3,#A+1024; R3 = address if the first word beyond A
LOOP:               ADD R1,(R2); register indirect through R2 to get operand
        ADD R2,#4     ; increment R2 by one word (4 bytes)
        CMP R2,R3     ; are we done yet?
        BLT LOOP      ; if R2 < R3, we are not done, so continue
```

Figure 119: Adding the elements of an array.

```
        MOV R1,#0     ; accumulate the OR in R1, initially 0
        MOV R2,#0     ; R2 = index, i, of current product: A[i] AND B[i]
        MOV R3,#4096; R3 = first index value not to use
LOOP:               MOV R4,A(R2); R4 = A[i]
        AND R4,B(R2); R4 = A[i] AND B[i]
        OR R1,R4      ; OR all the Boolean products into R1
        ADD R2,#4     ; i = i + 4 (step in units of 1 word = 4 bytes)
        CMP R2,R3     ; are we done yet?
        BLT LOOP      ; if R2 < R3, we are not done, so continue
```

Figure 120: Or-ing elements in an array.

- Base-index addressing - the memory address is computed by adding 2 registers, and optionally an offset:

```
LOOP: MOV R4, (R2 + R5)
      AND R4, (R2 + R6)
```

- Stack addressing is done using reverse Polish notation (see Figure 121).
- For branching instructions, any of the following addressing modes can be used:
  - register indirect addressing (but it could cause bugs hard to detect),
  - indexed mode (same problem),
  - offset from PC.

Addressing modes: examples

- Figure 122 illustrates the addressing modes supported by our example architectures.

Infix	Reverse Polish notation
A + B × C	A B C × +
A × B + C	A B × C +
A × B + C × D	A B × C D × +
(A + B) / (C – D)	A B + C D – /
A × B / C	A B × C /
((A + B) × C + D)/(E + F + G)	A B + C × D + E F + G + /

Figure 121: Reverse Polish notation correspondent to infix terms.

Addressing mode	Pentium II	UltraSPARC II	JVM
Immediate	×	×	×
Direct	×		
Register	×	×	
Register indirect	×		
Indexed	×	×	×
Based-indexed		×	
Stack			×

Figure 122: A comparison of supported addressing modes.

5.7 Instruction types

- Instruction types:
  - data movement instructions,
  - dyadic instructions,
  - monadic instructions,
  - comparison and conditional branches,
  - procedure calls instructions,
  - loop control,
  - I/O.
- "Data movement" or "data duplication" instructions - move data from one place to another.
- Two reasons for this:
  - assignment of values to variables (`A = B`),
  - copy data to stage it for efficient access and use.

- Four possible types of data movement instructions (not all of them possible on all machines):
  - memory-memory,
  - memory-register,
  - register-memory,
  - register-register.
- Data movement instructions may also indicate the amount of data moved:
  - one word,
  - less than a word (usually in increment of bytes),
  - multiple words.

Dyadic operations

- Combine two operands to produce a result.
- Typical dyadic operation:
  - arithmetic instructions (integer, floating point),
  - boolean instructions (not all possible boolean instructions are implemented, usually AND, OR, NOT, often XOR, NOR, NAND).
  - Special uses for AND, OR include extracting the bytes from words, by using byte masks (exercise: how?).

Monadic operations

- One operand.
- Typical monadic operations are shifting, rotating contents of the operands.
- Right shifts are often performed with sign extension (fill in the vacated positions with the sign).
- Shifts are used in efficient implementations of multiplication and division by powers of 2.
- Monadic and dyadic operations are often grouped by functionality and not the number of their arguments.

Comparisons and conditional branches

- Instructions to test data and alter the sequence of instructions to be executed, based on these results.
- This is usually performed as a sequence of two instructions:
  - test some condition,
  - is the condition is met, branch to some particular memory address.
- Typical conditions:
  - is a bit 0?
  - is a word 0?
  - do two sequences have the same size?
- Some machines set condition code bits that are used to indicate specific conditions: overflow, carry.
- Subtle points:
  - comparing 2 numbers (large negative and large positive can cause overflow when substracted),
  - ordering numbers ("is 011 greater than 100?") - decide whether they are signed or not.

Procedure call instructions

- Used to invoke a group of instructions that perform a certain task (procedure).
- Other names: subroutines (esp. in assembly languages), method (Java).
- When a procedure is finished, it has to return to the statement after the call.
- Therefore, the return address should be transmitted to the procedure, or stored somewhere:
  - fixed memory location (may cause problems with multiple procedure calls),
  - first word of the procedure, return by indirect branching (may lead to problems with recursive procedure),
  - register (again, problems with multiple procedure calls - not too many possible),
  - stack.

Loop control

- Instructions that allow the execution of a group of instructions a fixed number of times.
- All these involve a counter that is increased/decreased every time it goes through the loop, and it is tested for a condition - when the condition holds, the loop is terminated.
- Figure 123 illustrate possible loop instructions.

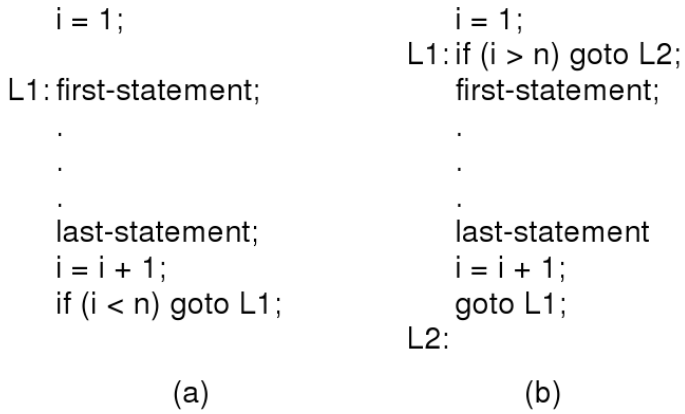


Figure 123: Looping with test (a) at the end and (b) test at the beginning.

I/O

- A large variety of I/O instructions are possible, but some of the most important are:
  - programmed I/O with busy waiting,
  - interrupt-driven I/O,
  - DMA I/O.
- Programmed I/O with busy waiting:
  - used in low-end microprocessors (embedded systems, real-time systems),
  - these usually have one input instruction and one output instruction,
  - each instruction selects one of the I/O devices, and a single character is transferred between the fixed register in the processor and the I/O/.

  - Figure 124 illustrates this type of I/O.
  - the disadvantage of programmed I/O: the CPU spends most of the time in a tight loop waiting for the I/O device to become ready (OK for washing machines CPUs).

```
public static void output_buffer(int buf[ ], int count) {
   // Output a block of data to the device
   int status, i, ready;
   for (i = 0; i < count; i++) {
      do {
         status = in(display_status_reg);// get status
         ready = (status << 7) & 0x01;// isolate ready bit
      } while (ready == 1);
      out(display_buffer_reg, buf[i]);
   }
}
```

Figure 124: A Java code fragment using programmed I/O.

- Interrupt-driven I/O:
  - every time a device becomes ready, an interrupt issues a signal,
  - the processor is free between interrupts,
  - interrupts can be expensive, especially if they are in great number (1 for each character).
- DMA I/O:
  - back to programmed I/O, but let somebody else do it (the DMA controller),
  - DMA controller: a chip with at least 4 registers (address, how many words are transferred, what device and what direction), see Figure 125,
  - DMA is not for free: cycle stealing - take the bus away from the CPU.

5.8 Flow of control

- Flow of control refers to the sequence in which instructions are executed dynamically, during program execution.

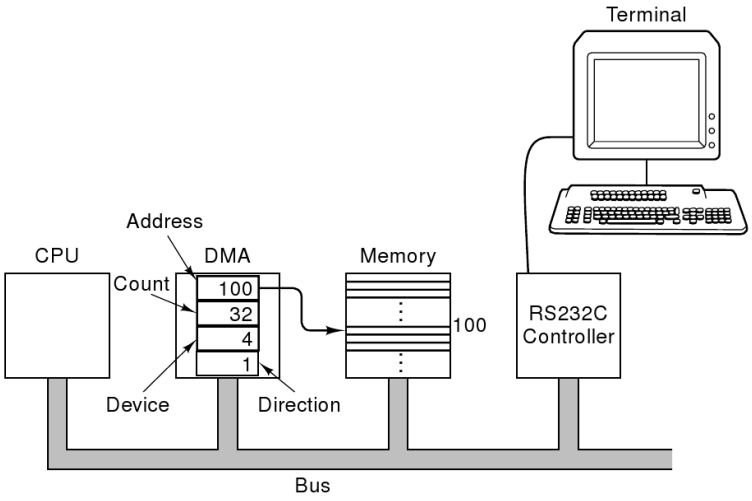


Figure 125: A system with a DMA controller.

- Figure 126 illustrates possible flows.
- Dijkstra (1968): "GO TO statement considered harmful" → the structured programming revolution.

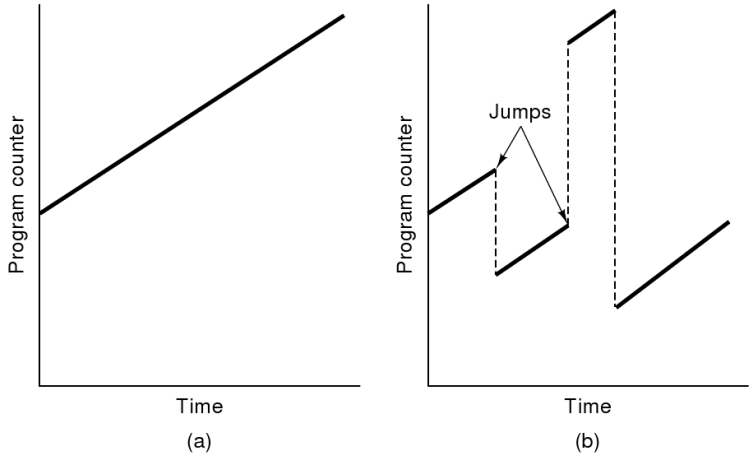


Figure 126: Program counter as a function of time (a) without branching, (b) with branching.

- Procedures are considered the most important technique for structuring programs.
- The flow of control comes back at the address following the caller's address (see Figure 127).
- Procedures can be seen as high level instructions.
- Interesting case - recursive procedures

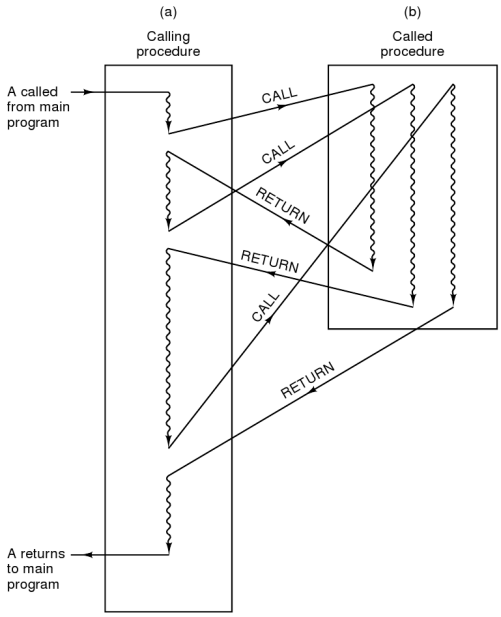


Figure 127: Procedure calls.

- Coroutines - procedures that run "in the same time" but do not start all over again when they are called (see Figure 128).
- Useful in simulating parallel processes.

- Traps are a kind of automatic procedure calls initiated by some condition caused by the program, usually an important but rarely occurring condition (e.g. overflow).
- Trap handlers are procedures that perform appropriate actions (e.g. error messages).

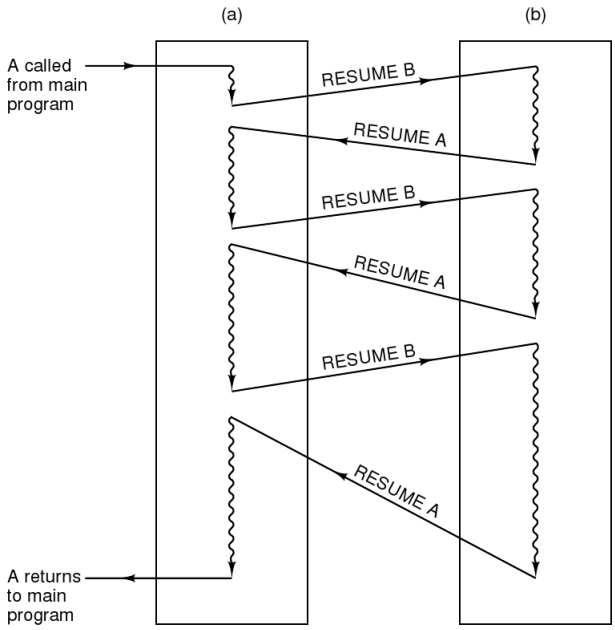


Figure 128: When a coroutine is resumed, execution starts from where it left off.

- Common conditions for traps: floating-point overflow, floating-point underflow, integer overflow, protection violation, undefined opcode, stack overflow, attempt to start inexistent I/O device, attempt to fetch a word from an odd numbered address, division by 0.
- Traps are synchronous with the program (they appear in the same spot every time the program runs with the same arguments).
- Interrupts are changes in the flow of control caused no by the running program, but by something else, usually some I/O device.
- Interrupts stop the running program and control is passed to interrupt handlers.
- Interrupts are asynchronous with the program (interrupts are, at best, only indirectly caused by the program).
- Interrupt transparency is important: when an interrupt happens, some actions are taken and code runs, but after, the computer should be returned to exactly the same state it had before the interrupt.
- In real life, interrupts may appear from many I/O devices in the same time

– first solution is for interrupt routines to disable any other interrupt, until they are finished (this is problematic for devices that cannot tolerate much delay),

– second solution is to attach priority levels to I/O devices, high for time-critical devices, see Figure 129.

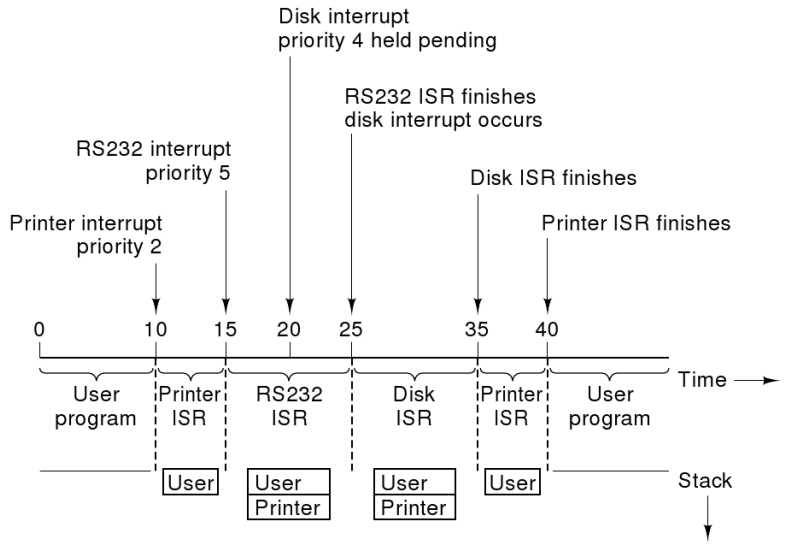


Figure 129: A machine with 3 I/O devices, a printer, a disk, a RS232 line, with priorities 2, 4, 5 respectively. (IRS - Interrupt service routine).

5.9 Example ISAs

Intel 32 bit ISA

- The Intel 32 bit integer ISA is illustrated in Figure 130.

UltraSPARC ISA

- Figure 131 illustrates the UltraSPARC integer instructions.

JVM ISA

- An overview of the JVM ISA is illustrated in Figure 132.

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOV DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract DST from SRC
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract DST & carry from SRC
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT ADDR	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source
DST = destination
# = shift/rotate count
LV = # locals

Figure 130: Intel 32 bit integer ISA.

5.10 Comparison of the Instruction Sets

- Intel 32 bit ISA:
  - classic two-address 32 bit CISC machine, with a long history, peculiar and irregular addressing modes, many instructions that reference memory,
  - driving factors:
    1. backward compatibility,
    2. backward compatibility,

Loads

LDSB ADDR,DST	Load signed byte (8 bits)
LDUB ADDR,DST	Load unsigned byte (8 bits)
LDSH ADDR,DST	Load signed halfword (16 bits)
LDUH ADDR,DST	Load unsigned halfword (16)
LDSW ADDR,DST	Load signed word (32 bits)
LDUW ADDR,DST	Load unsigned word (32 bits)
LDX ADDR,DST	Load extended (64-bits)

Stores

STB SRC,ADDR	Store byte (8 bits)
STH SRC,ADDR	Store halfword (16 bits)
STW SRC,ADDR	Store word (32 bits)
STX SRC,ADDR	Store extended (64 btis)

Arithmetic

ADD R1,S2,DST	Add
ADDCC "	Add and set icc
ADDC "	Add with carry
ADDCCC "	Add with carry and set icc
SUB R1,S2,DST	Subtract
SUBCC "	Subtract and set icc
SUBC "	Subtract with carry
SUBCCC "	Subtract with carry and set icc
MULX R1,S2,DST	Multiply
SDIVX R1,S2,DST	Signed divide
UDIVX R1,S2,DST	Unsigned divide
TADCC R1,S2,DST	Tagged add

Shifts/rotates

SLL R1,S2,DST	Shift left logical (64 bits)
SLLX R1,S2,DST	Shift left logical extended (64)
SRL R1,S2,DST	Shift right logical (32 bits)
SRLX R1,S2,DST	Shift right logical extended (64)
SRA R1,S2,DST	Shift right arithmetic (32 bits)
SRAX R1,S2,DST	Shift right arithmetic ext. (64)

Boolean

AND R1,S2,DST	Boolean AND
ANDCC "	Boolean AND and set icc
ANDN "	Boolean NAND
ANDNCC "	Boolean NAND and set icc
OR R1,S2,DST	Boolean OR
ORCC "	Boolean OR and set icc
ORN "	Boolean NOR
ORNCC "	Boolean NOR and set icc
XOR R1,S2,DST	Boolean XOR
XORCC "	Boolean XOR and set icc
XNOR "	Boolean EXCLUSIVE NOR
XNORCC "	Boolean EXCL. NOR and set icc

Transfer of control

BPcc ADDR	Branch with prediction
BPr SRC,ADDR	Branch on register
CALL ADDR	Call procedure
RETURN ADDR	Return from procedure
JMPL ADDR,DST	Jump and Link
SAVE R1,S2,DST	Advance register windows
RESTORE "	Restore register windows
Tcc CC,TRAP#	Trap on condition
PREFETCH FCN	Prefetch data from memory
LDSTUB ADDR,R	Atomic load/store
MEMBAR MASK	Memory barrier

Miscellaneous

SETHI CON,DST	Set bits 10 to 31
MOVcc CC,S2,DST	Move on condition
MOVr R1,S2,DST	Move on register
NOP	No operation
POPC S1,DST	Population count
RDCCR V,DST	Read condition code register
WRCCR R1,S2,V	Write condition code register
RDPC V,DST	Read program counter

SRC = source register
DST = destination register
R1 = source register
S2 = source: register or immediate
ADDR = memory address

TRAP# = trap number
FCN = function code
MASK = operation type
CON = constant
V = register designator

CC = condition code set
R =destination register

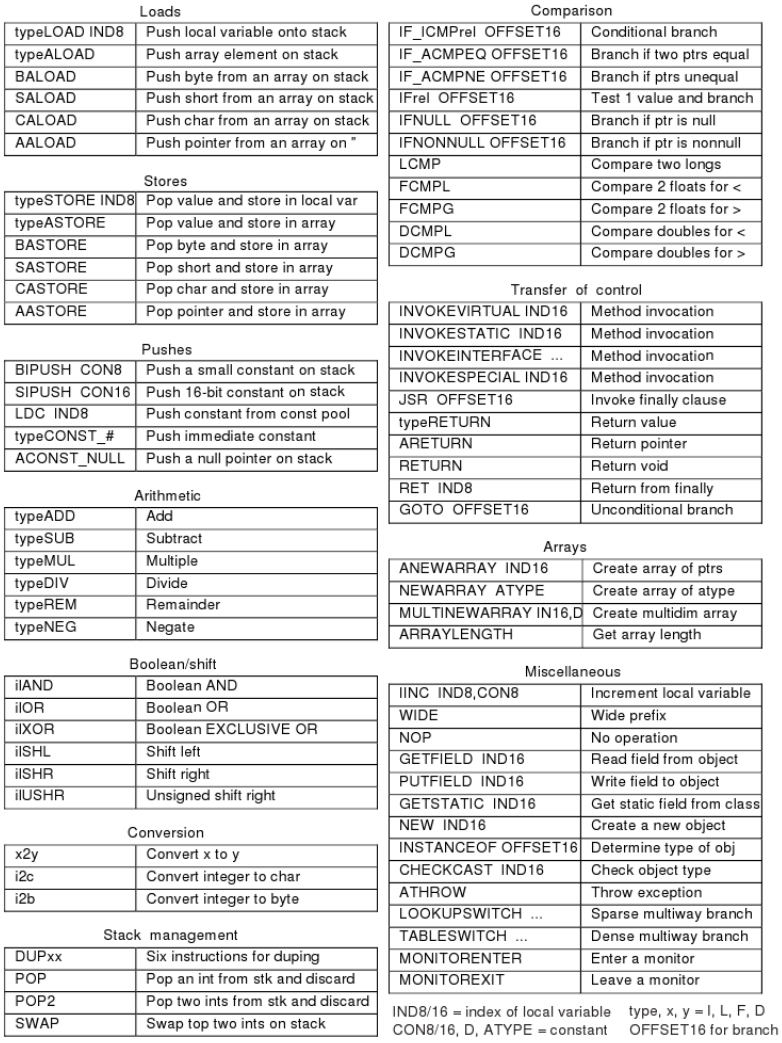
cc = condition
r = LZ,LEZ,Z,NZ,GZ,GEZ

Figure 131: UltraSPARC integer ISA.

3. backward compatibility.

   - compilers hard to write for the irregular instructions,
   - too few registers.

- UltraSPARC:
  - state-of-the-art ISA design,
  - 64 bit ISA, many registers, instructions (mostly 3 register operations),
  - instructions have the same size.
- JVM:



Loads

typeLOAD IND8	Push local variable onto stack
typeALOAD	Push array element on stack
BALOAD	Push byte from an array on stack
SALOAD	Push short from an array on stack
CALOAD	Push char from an array on stack
AALOAD	Push pointer from an array on "

Stores

typeSTORE IND8	Pop value and store in local var
typeASTORE	Pop value and store in array
BASTORE	Pop byte and store in array
SASTORE	Pop short and store in array
CASTORE	Pop char and store in array
AASTORE	Pop pointer and store in array

Pushes

BIPUSH CON8	Push a small constant on stack
SIPUSH CON16	Push 16-bit constant on stack
LDC IND8	Push constant from const pool
typeCONST_#	Push immediate constant
ACONST_NULL	Push a null pointer on stack

Arithmetic

typeADD	Add
typeSUB	Subtract
typeMUL	Multiple
typeDIV	Divide
typeREM	Remainder
typeNEG	Negate

Boolean/shift

iIAND	Boolean AND
iIOR	Boolean OR
iIXOR	Boolean EXCLUSIVE OR
iISHL	Shift left
iISHR	Shift right
iIUSHR	Unsigned shift right

Conversion

x2y	Convert x to y
i2c	Convert integer to char
i2b	Convert integer to byte

Stack management

DUPxx	Six instructions for duping
POP	Pop an int from stk and discard
POP2	Pop two ints from stk and discard
SWAP	Swap top two ints on stack

Comparison

IF_ICMPrel OFFSET16	Conditional branch
IF_ACMPEQ OFFSET16	Branch if two ptrs equal
IF_ACMPNE OFFSET16	Branch if ptrs unequal
IFrel OFFSET16	Test 1 value and branch
IFNULL OFFSET16	Branch if ptr is null
IFNONNULL OFFSET16	Branch if ptr is nonnull
LCMP	Compare two longs
FCMPL	Compare 2 floats for <
FCMPG	Compare 2 floats for >
DCMPL	Compare doubles for <
DCMPG	Compare doubles for >

Transfer of control

INVOKEVIRTUAL IND16	Method invocation
INVOKESTATIC IND16	Method invocation
INVOKEINTERFACE ...	Method invocation
INVOKESPECIAL IND16	Method invocation
JSR OFFSET16	Invoke finally clause
typeRETURN	Return value
ARETURN	Return pointer
RETURN	Return void
RET IND8	Return from finally
GOTO OFFSET16	Unconditional branch

Arrays

ANEWARRAY IND16	Create array of ptrs
NEWARRAY ATYPE	Create array of atype
MULTINEWARRAY IN16,D	Create multidim array
ARRAYLENGTH	Get array length

Miscellaneous

IINC IND8,CON8	Increment local variable
WIDE	Wide prefix
NOP	No operation
GETFIELD IND16	Read field from object
PUTFIELD IND16	Write field to object
GETSTATIC IND16	Get static field from class
NEW IND16	Create a new object
INSTANCEOF OFFSET16	Determine type of obj
CHECKCAST IND16	Check object type
ATHROW	Throw exception
LOOKUPSWITCH ...	Sparse multiway branch
TABLESWITCH ...	Dense multiway branch
MONITORENTER	Enter a monitor
MONITOREXIT	Leave a monitor

IND8/16 = index of local variable type, x, y = I, L, F, D
CON8/16, D, ATYPE = constant OFFSET16 for branch

Figure 132: The JVM ISA.

– designed initially for applets (small programs), short instructions (1.8 bytes average),

– 64 bit stack, instructions bundled together,

– the core of JVM is a deeply pipelined 3 register load/store RISC engine.

6 The Operating System Machine Level

- Operating system = a program that adds a variety of new instructions and features above and beyond the ISA level, usually implemented in software (but not necessarily so).
- Examples: Windows (XP/Vista/7), UNIX, GNU/Linux, OSX.
- The level that implements the operating system is the Operating System Machine Level.
- OSM includes most of the ISA instructions, but also adds others - system calls.
- The OSM level is always interpreted:
  - when a program executes an OSM instruction, the OS carries it out step by step,
  - when a program executes an ISA instruction, this is carried out directly by the underlying level.

Overview

- In the following, we give a brief description of:
  - virtual memory - a technique that makes the machine appear to have more memory than it actually has,
  - file I/O - higher level I/O operations (as compared to ISA I/O),
  - parallel processing - how multiple processes can be executed at the same time, how they communicate, synchronize.

6.1 Virtual Memory

- In the early days, computers had very little memory (Algol 60 was written for only 1024 words of memory).
- First solution to deal with this - overlays: divide the program such that each resulting piece fits into the main memory. One piece would be in the main memory, the rest in the secondary memory.
- The programmer had to divide the program and keep track of overlays (a process prone to errors).
- Overlays were used well into 1990s (DOS/Windows 3.1).
- Virtual memory (proposed by researchers in Manchester in 1961) is a technique that performs overlaying automatically.
- Virtual memory is available now to most computers.

- The key idea in the method is to separate the concepts of address space and memory locations (see Figure 133).

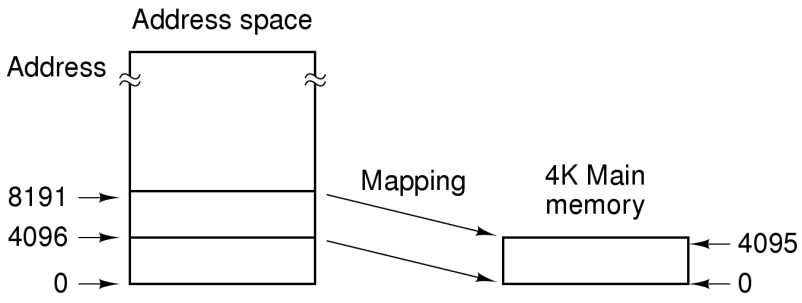


Figure 133: Virtual memory: mapping virtual addresses to memory locations.

Paging

- Paging is a technique for automatic overlaying.
- Pages are chunks of programs read from the disk.
- Common terms:
  - virtual address space: the address a program can refer to,
  - physical address space: the hardwired memory available on the machine,
  - memory map (page table): relates virtual addresses to physical spaces.
- Programs are written as if the whole virtual spacing is available.
- The paging mechanism is transparent: the programmer does not need to handle this explicitly.
- The memory management unit (MMU) is a device on the CPU chip or in its close proximity, that maps virtual addresses to their corresponding physical address.
- When a reference is made to an address that is not present in the memory, a page fault occurs, this is indicated as a trap and the operating system locates it in the secondary memory and brings it into the main memory.
- Paging models:
  - demand paging: load pages on demand, as they are needed,
  - working set model: load the most used pages for each program (the working set).

Segmentation

- The paging model is one-dimensional (one memory space).
- Typically, several memory spaces are needed (symbol table, source text, constant table, parse tree, stack call).
- Possible solutions within the one-dimensional models are not satisfactory:
  - preallocating (thus limiting) segment space is not practical,
  - handling explicitly the management of preallocated space overflow amounts to going back to overlays.
- Segmentation offers a solution to handle the multiple address spaces.
- Advantages of segmentation:
  - avoids recompilation (needed in the one-dimensional case),
  - facilitates communication between programs,
  - memory protection (not possible to write data over instructions).
- Figure 134 gives a comparison of paging and segmentation.

Consideration	Paging	Segmentation
Need the programmer be aware of it?	No	Yes
How many linear addresses spaces are there?	1	Many
Can virtual address space exceed memory size?	Yes	Yes
Can variable-sized tables be handled easily?	No	Yes
Why was the technique invented?	To simulate large memories	To provide multiple address spaces

Figure 134: A comparison of paging and segmentation.

6.2 Virtual I/O Instructions

- I/O instructions are one of the areas where the ISA and OS levels differ:
  - ISA I/O instructions are potentially dangerous, as they allow almost unrestricted access,
  - programming ISA I/O is tedious and complex.
- Files are an abstraction used in organizing virtual I/O:
  - a file consists of a sequence of bytes written to an I/O device,
  - any further structure of files is up to the application programs.

- File I/O is done by system calls for opening, reading, writing, closing files:
  - opening: the OS must locate the file on disk and bring it to memory,
  - reading: which file is to be read, pointer to a buffer where data should be placed, number of bytes to read,
  - after reading, the file is closed and the OS can use the file for another read/write.
- Mainframe operating systems have a more complicated concept of files: a file is a sequence of logical records, each with a well-defined structure.
- Virtual I/O instructions are implemented taking into account how files are organized and stored:
  - files can be allocated in consecutive sectors on the disk (simple to handle - CDROMS),
  - files can be allocated in nonconsecutive sectors (hard disks) - a file index keeps track of the correspondence between addresses and disk locations (UNIX) or the file is organized as a linked list (MS DOS, Windows 9x).
- Directories are further abstractions, used to group and organize files.
- System calls are provided for directory management:
  - creating a file and placing it in a directory,
  - deleting a file from a directory,
  - rename a file,
  - change the protection status of a file.
- In modern operating systems, directories are files themselves ($\rightarrow$ hierarchies of directories).

6.3 Virtual Instructions for Parallel Processes

- A process is an application program in execution.
- When an user starts a program, a new process is created, and the operating system manages its execution.
- Modern operating systems allow several processes to run in parallel:
  - by executing these processors on different processors (true parallelism),
  - by simulating parallel processing (see Figure 135).
- A process consists of:

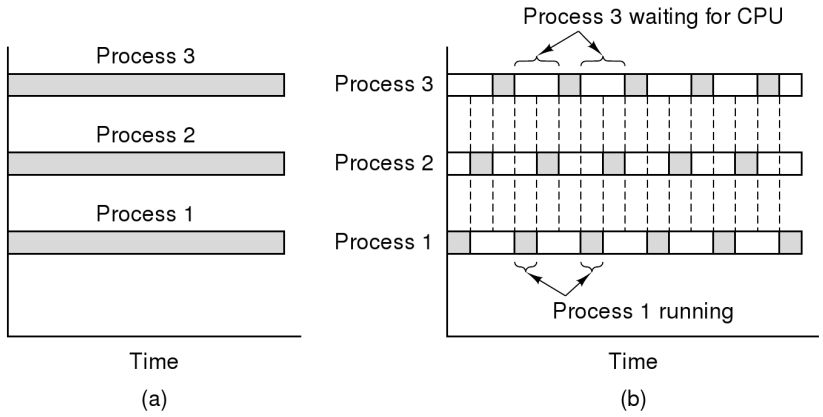


Figure 135: Process parallelism (a) True parallelism. (b) Simulated parallelism.

  - its executable code loaded from disk into the memory,
  - its stack, where program data is located,
  - its heap, where the program may allocate additional data,
  - its registers (including the program pointer and stack pointer),
  - additional information (e.g. user privileges).

- The possible states of a process can be:
  - executing: the CPU is executing instructions of the process,
  - ready: the process is ready for execution, but the CPU is executing some other process,
  - blocked: the process is waiting for some input (e.g. keyboard, data from some other process).

- The operating system must provide instructions for dynamic process creation.

- A process (parent) can control the process it created (child) - virtual instructions must be provided to stop, restart, terminate child processes.

- Synchronization of processes is an important task in any operating system, and various techniques are implemented (e.g. semaphores - more in the OS lecture).

- At any time, the operating system holds a pool of ready processes.

- Preemptive scheduling - the assignment of processes to simulate parallelism:

  - the executing process receives a time slice,
  - after the time slice has expired, the OS preempts the process,
  - the process is put into the ready pool and another is scheduled for execution,
  - rapid switching creates the illusion that the processes are active simultaneously.

- To request an OS service, a process performs a system call (trap):
  - a special processor instruction gives the control to the OS,
  - the OS takes data from registers and decides which services to perform,
  - the OS invokes the system service that interacts with the hardware device,
  - the OS returns control to the application.

6.4 Example Operating Systems

UNIX

- Figure 136 illustrates the types of system calls in UNIX, whereas Figure 137 gives the structure of a typical Unix system.

Category	Some examples
File management	Open, read, write, close, and lock files
Directory management	Create and delete directories; move files around
Process management	Spawn, terminate, trace, and signal processes
Memory management	Share memory among processes; protect pages
Getting/setting parameters	Get user, group, process ID; set priority
Dates and times	Set file access times; use interval timer; profile execution
Networking	Establish/accept connection; send/receive message
Miscellaneous	Enable accounting; manipulate disk quotas; reboot the system

Figure 136: A rough breakdown of UNIX calls.

Windows (NT)

- Figure 138 illustrates the structure of Windows (NT).

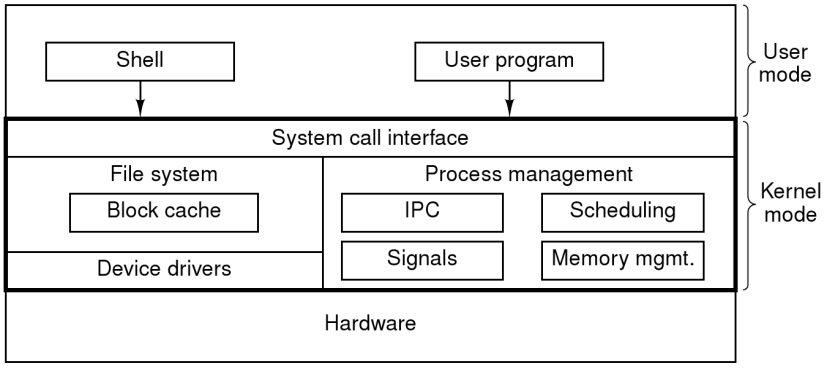


Figure 137: A typical UNIX system.

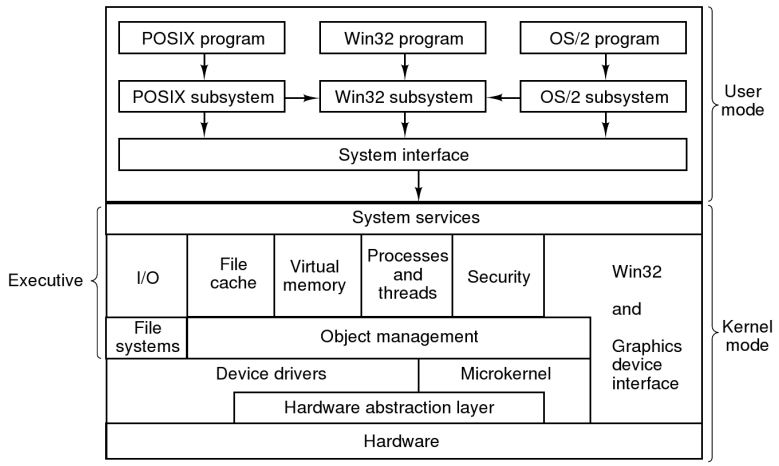


Figure 138: The structure of Windows (NT).

7 The Assembly Language Level

Overview

- The assembly language level is implemented by translation, and not by interpretation (like the underlying levels).
- The source is the assembly language program (sequence of text statements).
- The target is the machine language, each statement being translated into one machine instruction, thus generating an object file.
- The translator is the assembler, which performs the following tasks:
  - translate symbolic instruction names into numerical instruction codes,
  - translate register names into register numbers,
  - translate symbolic variables names into numerical memory locations.

Why use assembly language?

- An assembly language programmer has access to all the features of the target machine (e.g. parity bit registers).
- Assembly language programming is tedious, complex, slow (assembly programs are developed slower by a considerable factor compared to high level languages) and prone to error - why use it?
- Performance:
  - assembly language programs can be considerably faster than high level counterparts,
  - they can be much smaller (good for programming embedded applications),
  - they make better use of limited resources.
- Access to the machine:
  - the assembly language has complete access to the hardware,
  - it can implement device controllers, handle OS interrupts, etc.
- Figure 139 shows a small fragment of assembly code in various languages.

Label	Opcode	Operands	Comments
FORMULA:	MOV	EAX,I	; register EAX = I
	ADD	EAX,J	; register EAX = I + J
	MOV	N,EAX	; N = I + J
I	DW	3	; reserve 4 bytes initialized to 3
J	DW	4	; reserve 4 bytes initialized to 4
N	DW	0	; reserve 4 bytes initialized to 0

(a)

Label	Opcode	Operands	Comments
FORMULA	MOVE.L	I, D0	; register D0 = I
	ADD.L	J, D0	; register D0 = I + J
	MOVE.L	D0, N	; N = I + J
I	DC.L	3	; reserve 4 bytes initialized to 3
J	DC.L	4	; reserve 4 bytes initialized to 4
N	DC.L	0	; reserve 4 bytes initialized to 0

(b)

Label	Opcode	Operands	Comments
FORMULA:	SETHI	%HI(I),%R1	! R1 = high-order bits of the address of I
	LD	[%R1+%LO(I)],%R1	! R1 = I
	SETHI	%HI(J),%R2	! R2 = high-order bits of the address of J
	LD	[%R2+%LO(J)],%R2	! R2 = J
	NOP		! wait for J to arrive from memory
	ADD	%R1,%R2,%R2	! R2 = R1 + R2
	SETHI	%HI(N),%R1	! R1 = high-order bits of the address of N
	ST	%R2,[%R1+%LO(N)]	
I:	.WORD 3		! reserve 4 bytes initialized to 3
J:	.WORD 4		! reserve 4 bytes initialized to 4
N:	.WORD 0		! reserve 4 bytes initialized to 0

(c)

Figure 139: Implementing I+J = N (a) on Intel IA32, (b) Motorola 680x0 (c) UltraSPARC.

Pseudoinstructions

- An assembly language program can contain instructions to the assembler itself - pseudoinstructions (assembler directives).
- Examples (MASM - Microsoft Intel assembler):
  - `SEGMENT, ENDS` - start/end a segment,
  - `ALIGN` - control the alignment of the data,
  - `EQU` - give a symbolic name to an expression,
  - `DW` - allocate storage for 32 bit words,
  - `IF, ELSE, ENDIF` - conditional assembly:

```
WORDSIZE EQU 16
IF WORDSIZE EQU 16
 WSIZE: DW 32ELSE
 WSIZE: DW 16
ENDIF.
```

Macros

- A macro is a name for a code segment, that can be used over and over again (see Figure 140):
  - a macro definition gives a name to a piece of code,
  - a macro call inserts the macro intro the code,
  - macros can be parametrized.
- The use of macros eliminates the overhead of procedures.

```
MOV     EAX,P
MOV     EBX,Q
MOV     Q,EAX
MOV     P,EBX

MOV     EAX,R
MOV     EBX,S
MOV     S,EAX
MOV     R,EBX
```

(a)

```
CHANGE  MACRO P1, P2
        MOV EAX,P1
        MOV EBX,P2
        MOV P2,EAX
        MOV P1,EBX
        ENDM

        CHANGE P, Q

        CHANGE R, S
```

(b)

Figure 140: Swapping (a) without macros, (b) with macros.

The assembly process

- The natural way to process an assembly language program is for the assembler to read one instruction and translate it into a machine instruction:
  - this leads to problems with conditional jump instructions (forward references),
  - solution: two-pass assemblers:
    - pass one: collect symbol information (labels) into the symbol table,
    - pass two: each statement is read, assembled, output (information has to be given to the linker, for linking up procedures).
  - the symbol table can be organized in various ways, to allow fast retrieval:
    - array of pairs,
    - sorted array,
    - hash coding.

- In real life, most programs consist of more than one procedure.
- The procedures must be linked together (by the linker).
- The complete translation of a source program requires two steps:
  1. assembly (compilation) of the source procedures,
  2. linking of the object modules.
- Figure 141 illustrates the translation process for assembly programs.

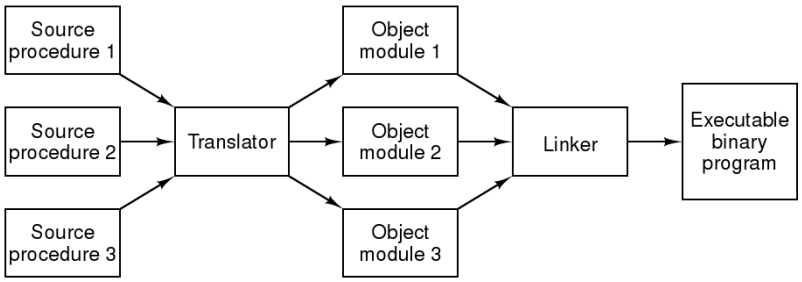


Figure 141: From source(s) to executable: the assembly process.

- Each module contains: entry point table (list of symbols), external reference table (list of external symbols used), relocation dictionary (addresses in the program that need to be relocated),
- The linker must:
  1. collect the table of modules,
  2. assign start addresses to modules,
  3. find all memory instructions and
  4. procedure calls instructions

  to linearize the program (with respect to memory locations).
- Dynamic linking occurs during the execution:
  - modules may be used by multiple processes,
  - implementation in modern operating systems:
    - DLL (Dynamic Link Libraries) in Windows,
    - shared library in UNIX.
- The use of dynamic linking reduces the size of the program file.

References

[East, 1990] East, I. (1990). Computer Architecture and Organization. Pitman Publishing, London.

[Harris and Harris, 2007] Harris, D. and Harris, S. (2007). Digital Design and Computer Architecture. Elsevier.

[Hennessy and Patterson, 2012] Hennessy, J. L. and Patterson, D. A. (2012). Computer Architecture: A Quantitative Approach. Elsevier, fifth edition edition.

[Hsu, 2001] Hsu, J. Y. (2001). Computer Architecture: Software Aspects, Coding, and Hardware. CRC Press.

[Tanenbaum, 2005] Tanenbaum, A. S. (2005). Structured Computer Organization. Prentice Hall.