Microprocessor and Microcontroller Architecture
Von Neumann Architecture

**Stored-Program Digital Computer**

- Digital computation in **ALU**
- **Programmable** via set of standard instructions
- Internal **storage** of data
- Internal **storage** of program
- Automatic **Input/Output**
- Automatic **sequencing** of instruction execution by decoder/controller

**Von Neumann Architecture**

Data and instructions stored in a single memory unit

**Harvard Architecture**

Data and instructions stored in a separate memory units
## Memory Hierarchy

### Levels of data / memory storage

<table>
<thead>
<tr>
<th>Data and instructions of &quot;all&quot; programs</th>
<th>&quot;All&quot; data and instructions of running programs</th>
<th>Copy small section of Main Memory for faster access</th>
<th>Fastest access to small amount of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access by OS call Width = allocation unit</td>
<td>Access by address Width = byte</td>
<td>Access by address Width = byte</td>
<td>Access by name Width = word</td>
</tr>
<tr>
<td>(\mu P): External (\mu C): External</td>
<td>(\mu P): External (\mu C): Internal</td>
<td>(\mu P): Internal (\mu C): —</td>
<td>(\mu P): Internal (\mu C): Virtual</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Long Term Storage**
- **Main Memory (RAM)**
- **Cache**
- **Register**

**Flow of data:**
- **All Files and Data**
- **Running Programs and Data**
- **Next Few Instructions and Data**
- **Current Data**
μP Subsystems

Microprocessor
- register memory
- Arithmetic Logic Unit (ALU)
- processor control

Cache Memory Unit
- cache memory
- cache control

Main Memory Unit
- main memory
- memory control

I/O System
- long-term storage
- network
- input
- output

Bus Controller

Processor package
μC Subsystems

Microprocessor

- data memory
- instruction memory
- processor control
- Arithmetic Logic Unit (ALU)

Controller package

- long-term storage
- I/O devices
- input
- output
- timers
Instruction Set Architecture

**General instruction — instance of data structure**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operand</th>
<th>Operand</th>
<th>...</th>
<th>Operand</th>
</tr>
</thead>
</table>

**Instruction set**

Range of data structure for

- **Operation** \( \in \{ \text{legal actions} \}\)
- **Operand** \( \in \{ \text{legal addressing modes} \}\)

**Machine instruction**

Binary code for processing by hardware

**Assembly instruction**

User-friendly form of machine instruction

Binary code \(\rightarrow\) words

**Typical instruction**

Machine: \(0x82E31F2B\)

Assembly: \texttt{ADD destination, source_1, source_2}\n
Definition: \texttt{destination} \(\leftarrow\) \texttt{source_1 + source_2}\)
General Operations

**Data transfer**

Load \( r \leftarrow m \), store \( m \leftarrow r \), move \( r/m \leftarrow r/m \), convert data types

**Arithmetic/ Logical (ALU)**

Integer arithmetic (\( + - \times \div \) compare shift) and logical (AND, OR, NOR, XOR)

**Decimal**

Integer arithmetic on decimal numbers

**Floating point (FPU)**

Floating point arithmetic (\( + - \times \div \text{sqrt trig exp} \ldots \))

**String**

String move, string compare, string search

**Control**

Conditional and unconditional branch, call/return, trap

**Operating System**

System calls, virtual memory management instructions

**Graphics**

Pixel operations, compression/decompression operations
Addressing Modes

Formal specification

Immediate (IMM)

Constant = literal = numerical value coded into instruction

Register operands

register name = a \( \mu P \) storage location

\[ \text{REGS[register name]} = \text{data stored in register} \]

\[ \text{REGS[R3]} = \text{data stored in register} \quad R3 = 11223340 \]

Memory operands

address = a memory storage location

\[ \text{MEM[address]} = \text{data stored in memory} \]

\[ \text{MEM[11223344]} = \text{data stored at address} \quad 11223344 = 45 \]

Effective Address (EA) — pointer arithmetic

\[ \text{REGS[R3]} \leftarrow &(\text{variable}) \]

\[ \text{MEM[REGS[R3]+4]} = *(&(\text{variable})+4) = *(\text{REGS[R3]}+4) \]

\[ = *(11223340+4) = 45 \]
### General Addressing Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Syntax</th>
<th>Memory Access</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>R3</td>
<td>Regs[R3]</td>
<td>Register data</td>
</tr>
<tr>
<td>Immediate</td>
<td>#3</td>
<td>3</td>
<td>Constant</td>
</tr>
<tr>
<td>Direct (absolute)</td>
<td>(1001)</td>
<td>Mem[1001]</td>
<td>Static data</td>
</tr>
<tr>
<td>Register deferred</td>
<td>(R1)</td>
<td>Mem[Regs[R1]]</td>
<td>Pointer</td>
</tr>
<tr>
<td>Displacement</td>
<td>100 (R1)</td>
<td>Mem[100+Regs[R1]]</td>
<td>Local variable</td>
</tr>
<tr>
<td>Indexed</td>
<td>(R1 + R2)</td>
<td>Mem[Regs[R1]+Regs[R2]]</td>
<td>Array addressing</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>@ (R3)</td>
<td>Mem[Mem[Regs[R3]]]</td>
<td>Pointer to pointer</td>
</tr>
<tr>
<td>Auto Increment</td>
<td>(R2) +</td>
<td>Mem[Regs[R2]]</td>
<td>Stack access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Regs[R2] ← Regs[R2]+d</td>
<td></td>
</tr>
<tr>
<td>Auto Decrement</td>
<td>-(R2)</td>
<td>Regs[R2] ← Regs[R2]−d</td>
<td>Stack access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mem[Regs[R2]]</td>
<td></td>
</tr>
<tr>
<td>Scaled</td>
<td>100 (R2) [R3]</td>
<td>Mem[100+Regs[R2]+Regs[R3]*d]</td>
<td>Indexing arrays</td>
</tr>
<tr>
<td>PC-relative</td>
<td>(PC)</td>
<td>Mem[PC+value]</td>
<td>Load instruction to data register</td>
</tr>
<tr>
<td>PC-relative deferred</td>
<td>1001 (PC)</td>
<td>Mem[PC+Mem[1001]]</td>
<td>Load instruction to data register</td>
</tr>
</tbody>
</table>
Read Only Memory (ROM)

Non-volatile memory

- Permanent configuration data
- Device initialization code (FIRMWARE)
- Basic Input/Output System (BIOS)

ROM technologies

- OTP (One-Time Programmable)
  - Cannot be changed after writing
- EPROM (Erasable Programmable Read-Only Memory)
  - Glass window allows ultraviolet light to erase device
- EEPROM (Electrical Erasable Programmable Read-Only Memory)
  - Can write electrically without prior erase

Flash

Organized as blocks

<table>
<thead>
<tr>
<th>Read</th>
<th>Write</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td></td>
<td>Read any bit in any block</td>
</tr>
<tr>
<td>Write</td>
<td>0 → 1</td>
<td>Overwrite and bit in any block</td>
</tr>
<tr>
<td></td>
<td>1 → 0</td>
<td>Copy block → delete entire block (write 0) → overwrite 1-bits</td>
</tr>
</tbody>
</table>
Register-Memory Model

Operands can be stored in any **REGISTER** or **MEMORY** location

```
Z = X + Y → load R1, X
    add R1, R1, Y
    store Z, R1
```

Easier to program

Register-Register Model

**MEMORY** operands must be loaded to a **REGISTER**

Also called **LOAD-STORE MODEL**

```
Z = X + Y → load R1, X
    load R2, Y
    add R1, R1, R2
    store Z, R1
```

Easier to implement in hardware

Statistics → most loaded operands used more than once
Running Machine Language Program

Program list

Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Instruction 6
...

μP

Fetches next instruction in list
Decodes fetched instruction
Executes decoded instruction

Instructions in Flat Memory

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>byte</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>byte</td>
<td>A+1</td>
</tr>
<tr>
<td>Instruction 2</td>
<td>byte</td>
<td>A+2</td>
</tr>
<tr>
<td></td>
<td>byte</td>
<td>A+3</td>
</tr>
<tr>
<td>Instruction 3</td>
<td>byte</td>
<td>A+4</td>
</tr>
<tr>
<td></td>
<td>byte</td>
<td>A+5</td>
</tr>
<tr>
<td>Instruction 4</td>
<td>byte</td>
<td>A+6</td>
</tr>
<tr>
<td></td>
<td>byte</td>
<td>A+7</td>
</tr>
<tr>
<td>Instruction 5</td>
<td>byte</td>
<td>A+8</td>
</tr>
<tr>
<td></td>
<td>byte</td>
<td>A+9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A+10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A+11</td>
</tr>
</tbody>
</table>
Modular Programming (Main + Functions)

Build program from separate source files (modules)

Each source module edited in a separate file

Compile / Assemble source files into separate object files

Object code is machine code with symbolic external references

Link object files together

Create executable file

Easier to design, read and understand programs

Write most modules in high level language
Write critical sections in assembly language
Write, debug, and change modules independently
Computer Design Before 1990

Limitations

- Memory = expensive
  - RAM ~ $5000/MB wholesale in 1977
- Compiler = bad
  - Bad error messaging
  - Weak optimization
  - Efficient code ⇒ write / optimize in assembly language

Implications

- Complex Instruction Set Computer (CISC)
- Easier assembly programming
  - Closer to high level language
- Powerful assembly language
  - > 300 instructions
  - > 12 addressing modes
  - > 10 data types
- Powerful instructions ⇒ fewer instructions ⇒ less memory
Machine Language Instruction

```
SUB R1, R2, 100 (R3)
```

Microcode Instruction Sequence (Microprogram)

- \( \text{ALU}_\text{IN} \leftarrow R3 \)
- \( \text{ALU} \leftarrow 100 \)
- \( \text{ADD} \)
- \( \text{MAR} \leftarrow \text{OUT} \)
- \( \text{READ} \)
- \( \text{ALU}_\text{IN} \leftarrow \text{MDR} \)
- \( \text{ALU} \leftarrow R2 \)
- \( \text{SUB} \)
- \( R1 \leftarrow \text{OUT} \)
Run Time and Clock Cycles

μP is timed by periodic signal called a clock

Clock Cycle time is measured in seconds per cycle

Instruction requires 1 or more clock cycles to process
Clock Rate is cycles per second = Hz (Hertz)

Run time = clock cycles to run program $\times$ seconds per clock cycles

$= \frac{\text{clock cycles to run program}}{\text{clock cycles per second}}$

Higher clock rate $\Rightarrow$ shorter run time
More clock cycles (at constant clock rate) $\Rightarrow$ longer run time

Clock Cycles Per Instruction (CPI) = lines of microcode
CISC Limitations

**Complex microcode**

Many instruction types ⇒ many microcode sequences
Complex operations ⇒ complex decoding and sequencing

**Central bus organization**

Permits atomic microcode operations
System bus ⇒ bottleneck
  - Microcode operations execute one-at-a-time
  - Machine instructions execute one-at-a-time
Microcode ⇒ several clock cycles to execute machine instruction

**Memory access**

Instruction length
  - Non-uniform
  - Depends on operation complexity
Multiple clock cycles to load instruction
Computer Design Since 1990

Technological developments

Price of RAM
$5000 / MByte (1975) → $5 / MByte (1990) → $0.01 / MByte (2012)

Compilers
Powerful + efficient + optimized

Reduced Instruction Set Computer (RISC)

Speed up most common operations

Fewer machine instructions with uniform instruction length (in bytes)
Ignore performance degradation to other operations

Simpler hardware design
No microcode
No system bus

All processors today use RISC technology

Pure RISC (PowerPC, Sparc, MIPS, ARM, Arduino, PIC, …)
RISC technology for CISC language (Pentium II – 4, Centrino, Core)
Explicitly parallel RISC (Intel Itanium, IBM mainframes)
## Typical RISC Instructions

<table>
<thead>
<tr>
<th>Class</th>
<th>Instruction</th>
<th>Operands</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transfer</td>
<td>Load</td>
<td>Integer / Float</td>
<td>R1 ← [R2 + offset]</td>
</tr>
<tr>
<td></td>
<td>Store</td>
<td></td>
<td>[R2 + offset] ← R1</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic</td>
<td>Register/Register</td>
<td>R1 ← R2 + R3</td>
</tr>
<tr>
<td></td>
<td>+ − × ÷</td>
<td>Register/Immediate</td>
<td>R1 ← R2 − Imm</td>
</tr>
<tr>
<td></td>
<td>Logic</td>
<td></td>
<td>R1 ← R2 ∧ R3</td>
</tr>
<tr>
<td></td>
<td>∧ ∨ ⊕</td>
<td></td>
<td>R1 ← R2 ∨ Imm</td>
</tr>
<tr>
<td></td>
<td>Test &amp; Set</td>
<td>Register/Register</td>
<td>R1 ← (R2 &gt; R3)</td>
</tr>
<tr>
<td>Float</td>
<td>+ − × ÷</td>
<td>Register/Register</td>
<td>F1 ← F2 × F3</td>
</tr>
<tr>
<td></td>
<td>Test &amp; Set</td>
<td></td>
<td>F1 ← (F2 ≠ F3)</td>
</tr>
<tr>
<td>Control</td>
<td>Branch</td>
<td>Immediate</td>
<td>PC ← PC + Imm</td>
</tr>
<tr>
<td></td>
<td>Branch Register</td>
<td>Register</td>
<td>PC ← R1</td>
</tr>
<tr>
<td></td>
<td>Branch &amp; Link</td>
<td>Register/Immediate</td>
<td>R31 ← PC</td>
</tr>
<tr>
<td></td>
<td>Conditional Branch</td>
<td></td>
<td>PC ← PC + [Imm * (F2 = 0)]</td>
</tr>
</tbody>
</table>
Typical RISC Pipeline

Stage 1
IF
Instruction Fetch

Stage 2
ID
Instruction Decode

Stage 3
EX
Execute

Stage 4
MEM
Data Memory Access

Stage 5
WB
Write Back

Address
Instruction Memory

Instruction

Address
Data Memory

Data

clock cycle

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>I₁</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I₂</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I₃</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I₄</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I₅</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I₆</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I₇</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I₈</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Microcontroller (μC) versus Microprocessor (μP)

Microprocessor (μP) application

General purpose computer
  Access + process data → data output
Computational power
  Programming generality
  Execution speed
Multiple processors for parallel processing
  Each μP handles thread

Microcontroller (μC) application

Embedded system
  Control external hardware operations
Cost efficiency
  Small number of program tasks stored in permanent memory
  Lowest possible cost
Multiple controllers for concurrent control problems
  Each μC applied to small group of tasks
Embedded-Processor Core

**Programmable Logic Devices (PLD/ FPGA)**

- Generic programmable integrated circuits
- Large array of digital circuit blocks
- User defines logic blocks
  - Truth tables
  - Boolean functions
  - Karnaugh diagrams
- User defines connections on programmable routing matrix
- Design copied to ASIC for manufacture
  - ASIC — application specific integrated circuit

**Embedded-processor core**

- μP or μC available on chip as programmable logic block
- Large embedded system designed on single chip
  - μP or μC works with other digital system blocks
- System on chip (SoC)
Choosing a Microcontroller — Generic Requirements

Optimum device for given application

Device family
  Uniform ISA
  Different hardware resources

Internal resources

Interrupts
  Type + number of I/O lines (analog and digital)
  Size of program and data memory

Space optimization

Smallest footprint at reasonable cost

Low power consumption

Battery powers applications using microcontrollers
  Sleep state while microcontroller idle

Copy protection

Stored program protected against user reading
Microcontroller Components

**Microprocessor core**
Typically RISC-type \( \mu \)P

**Memory**
-ROM holds program (FIRMWARE)
-RAM / registers for data + configuration
-Usually RAM < ROM

**Timers**
-Time internal / external events
-Watchdog — timeout resets system if code loop fails

**Controller I/ O**
-Interrupt controller — external event grabs processor attention
-Analog ↔ digital converters (A/D and D/A)
-Digital signal processor (DSP)
-Serial ↔ parallel converters (UART)

**Oscillator**
-Generates clock signal to synchronize all internal operations
Special Purpose Registers

**Instruction register**
Holds executing instruction

**Program counter**
Points to next instruction

**Accumulator**
Associated with ALU operations
One operand must be in ACC
Result stored in ACC

**Status register (flags)**
Set configuration
Results of ALU operations

**Data address register (DAR)**
Stores data memory addresses

**Stack pointer**
Points to last element pushed to stack
Reset

Initializes microcontroller

Sets PC to preset value (init address)
Microcontroller starts executing commands from init address

Causes of reset

Power up
Controller resets at startup

Manual reset
Press reset button

Power-glitch reset
Detect spike on power supply

Brown-out reset
Input voltage drops below threshold

Watchdog timer (WDT)
Power Consumption

**Low power consumption**
Most microcontroller applications on battery power

**Low power chip technology**
Complementary metal-oxide semiconductor (CMOS)

**Clock frequency**
Power consumed only on logic transition $1 \leftrightarrow 0$
Higher clock frequency $\Rightarrow$ more transitions /second $\Rightarrow$ more power

**Supply voltage**
Higher supply voltage $\Rightarrow$ faster + higher power

**Sleep state**
Stop clock $\Rightarrow$ 0 transitions /second $\Rightarrow$ no power
Leave low-power mode by external interrupt or reset
  - Key press
  - Interrupt
# Common Microcontrollers

<table>
<thead>
<tr>
<th>Applied Micro (was IBM)</th>
<th>PowerPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atmel</td>
<td>AVR family</td>
</tr>
<tr>
<td></td>
<td>AT89 family (Intel 8051 architecture)</td>
</tr>
<tr>
<td></td>
<td>AT91 family (ARM architecture)</td>
</tr>
<tr>
<td>Freescale</td>
<td>68HC00 family</td>
</tr>
<tr>
<td></td>
<td>DSP56800</td>
</tr>
<tr>
<td></td>
<td>MPC family</td>
</tr>
<tr>
<td>Intel</td>
<td>MCS-48 (8048 family)</td>
</tr>
<tr>
<td></td>
<td>MCS-51 (8051 family)</td>
</tr>
<tr>
<td></td>
<td>MCS-96 (8096 family)</td>
</tr>
<tr>
<td>Microchip Technology</td>
<td>PIC families</td>
</tr>
<tr>
<td>National Semiconductor</td>
<td>COP families</td>
</tr>
<tr>
<td>Sony</td>
<td>SPC families</td>
</tr>
<tr>
<td>STMicroelectronics</td>
<td>ST families</td>
</tr>
<tr>
<td>Texas Instruments</td>
<td>TMS families</td>
</tr>
<tr>
<td>Toshiba</td>
<td>TLCS families</td>
</tr>
<tr>
<td>Zilog</td>
<td>eZ8/80/16 families</td>
</tr>
</tbody>
</table>