Program Analysis

Lecture 02: *Machine Language*
Winter term 2011/2012

Prof. Thorsten Holz

Announcements

- Moodle info
  - http://moodle.ruhr-uni-bochum.de/course/view.php?id=689
    - Passwort: `PA2011`
- Introduction to OllyDbg and IDA Pro next week after the lecture

Outline

- Machine language / assembler
- Hardware basics
- Intel x86 architecture
  - Registers
  - Instruction set
  - Examples

Machine Language

Machine Language

- *Machine code* is sequence of elementary *machine instructions* (short: *instructions*)
- CPU works directly on machine code
- *Machine language* is textual representation of this machine code
- Set of all machine instructions is called *instructions set architecture* (ISA)
- Each architecture has its own machine language and distinct byte patterns (x86 vs. Intel/AMD)

Assembler

- Machine code is (almost) unreadable for humans (only bit sequence, e.g., 10110000 01100001)
- Present valid bit sequences as *mnemonics* and digits (e.g., `mov al, 61h`)
- *Assembler code* can be understood by humans
  - Assembler translates (almost) directly into machine code
  - Small differences between assembler and machines code (e.g., format of digits, comments)

Assembler

- Machine code is (almost) unreadable for humans (only bit sequence, e.g., 10110000 01100001)
- Present valid bit sequences as *mnemonics* and digits (e.g., `mov al, 61h`)

⇒ *Lowest layer for us: assembler*

- Assembler translates (almost) directly into machine code
- Small differences between assembler and machines code (e.g., format of digits, comments)

Example


```
.text:01005569 ; ---------------------------------------------------------------
.text:01005569
.text:01005569 loc_1005569:                            ; CODE XREF: LoadFile(x,x)+3E1j
.text:01005569                 mov esi, edi
.text:0100556B
.text:0100556B loc_100556B:                            ; CODE XREF: LoadFile(x,x)+3DAj
.text:0100556B                 lea ecx, [ebx+ebx]
.text:0100556E                 mov edi, eax
.text:01005570                 mov eax, ecx
.text:01005572                 shr ecx, 2
.text:01005575                 rep movsd
.text:01005577                 mov ecx, eax
.text:01005579                 and ecx, 3
.text:0100557C                 rep movsb
.text:0100557E                 mov edi, [ebp+lpBaseAddress]
.text:01005584                 mov esi, ds:__imp__SendMessageW@16
.text:0100558A                 jmp short loc_10055B0
.text:0100558C ; ---------------------------------------------------------------
```

Example II


```
ROM:00000058                STMFD   SP!, {R4-R11,LR}
ROM:0000005C                MOV     R11, #0x60000000
ROM:00000060                LDR     R9, [R11,#0x14]
ROM:00000064                SUB     SP, SP, #0x1C
ROM:00000068                MOV     R5, R0
ROM:0000006C                MOV     R6, R1
ROM:00000070                CMP     R1, #4
ROM:00000074                MOVNE   R7, #0x2000
ROM:00000078                BNE     loc_90
ROM:0000007C                BIC     R0, R9, #0x3000
ROM:00000080                ORR     R0, R0, #0x2000
ROM:00000084                ORR     R0, R0, #1
ROM:00000088                MOV     R7, #0x800
ROM:0000008C                STR     R0, [R11,#0x14]
ROM:00000090
ROM:00000090 loc_90                                 ; CODE XREF: sub_58+20j
ROM:00000090                LDR     R0, =unk_12D8
ROM:00000094                LDR     R0, [R0]
```

Example III


```
; CC5X Version 3.4E, Copyright (c) B Knudsen Data
; C compiler for the PICmicro family

   processor  16F54
   radix  DEC

i           EQU   0x07
j           EQU   0x08

test
    MOVWF i
    MOVF  i,W
    MOVWF j
    RETLW 0
main
    MOVLW 5
    CALL  test
    SLEEP

    ORG 0x01FF
    GOTO main
    END
```

What is Different?


```java
// Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
    public void paintComponent(Graphics g) {
        g.drawString("Hello, world!", 65, 95);
    }
}
```

```python
links, rechts = [], [] # Leere Listen links und rechts
pivotelement = liste.pop() # Das letzte Element aus der Liste nehmen
for element in liste: # Die verkürzte Liste durchlaufen
    if element < pivotelement:
        links.append(element) # wenn < dann an linke Liste anhängen
    else:
        rechts.append(element) # wenn nicht < (also >=) dann an rechte Liste anhängen
```

```cpp
int main() {
   char *buf;
   try {
      buf = new char[512];
      if( buf == 0 )
         throw "Memory allocation failure!";
   }
   catch( char * str ) {
      cout << "Exception raised: " << str << '\n';
   }
}
```

What is Different?


```
printf("%u\n", 13 * i + f(1));
```

What is Different?


```
printf("%u\n", 13 * i + f(1));
```

```
mov     esi, [ebp+var_4]
imul    esi, 0Dh
push    1
call    00401000
add     esp, 4
add     esi, eax
push    esi
push    004020F4
call    [0040209C]
```

Bytecode

- Interpreted languages (e.g., Python, Java, ActionScript) are translated into *bytecode*
- This bytecode is then executed by an interpreter ("processor for language")
- Assembler code != bytecode
- Many optimizations possible, e.g., *JIT compiler*

Why Machine Language?

- Source code in higher-level language is translated into machine code (task of the compiler)
- Source code is often not available, especially in the case of malicious software (e.g., bots)
- Analysis of machine code and also understanding of the underlying architecture and concepts is important

Why Machine Language?

- Quote from Donald Knuth:

Some people say that having machine language, at all, was the great mistake that I made. I really don't think you can write a book for serious computer programmers unless you are able to discuss low-level detail.

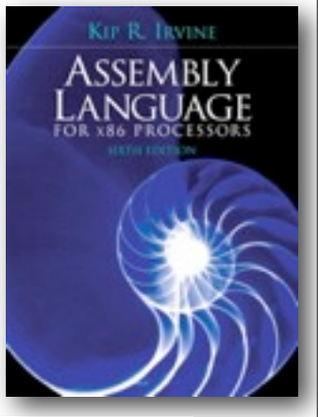
- Understanding machine language helps to understand inner working and concepts of computer systems
- Low-level knowledge leads to a better understanding of certain high-level concepts

Background Reading

- Intel64 and IA-32 Manual Vol. 1-3
- AMD Developer Guides & Manuals
- Both are freely available as PDF (link in Moodle)

- Very detailed description of the architecture and all of its features

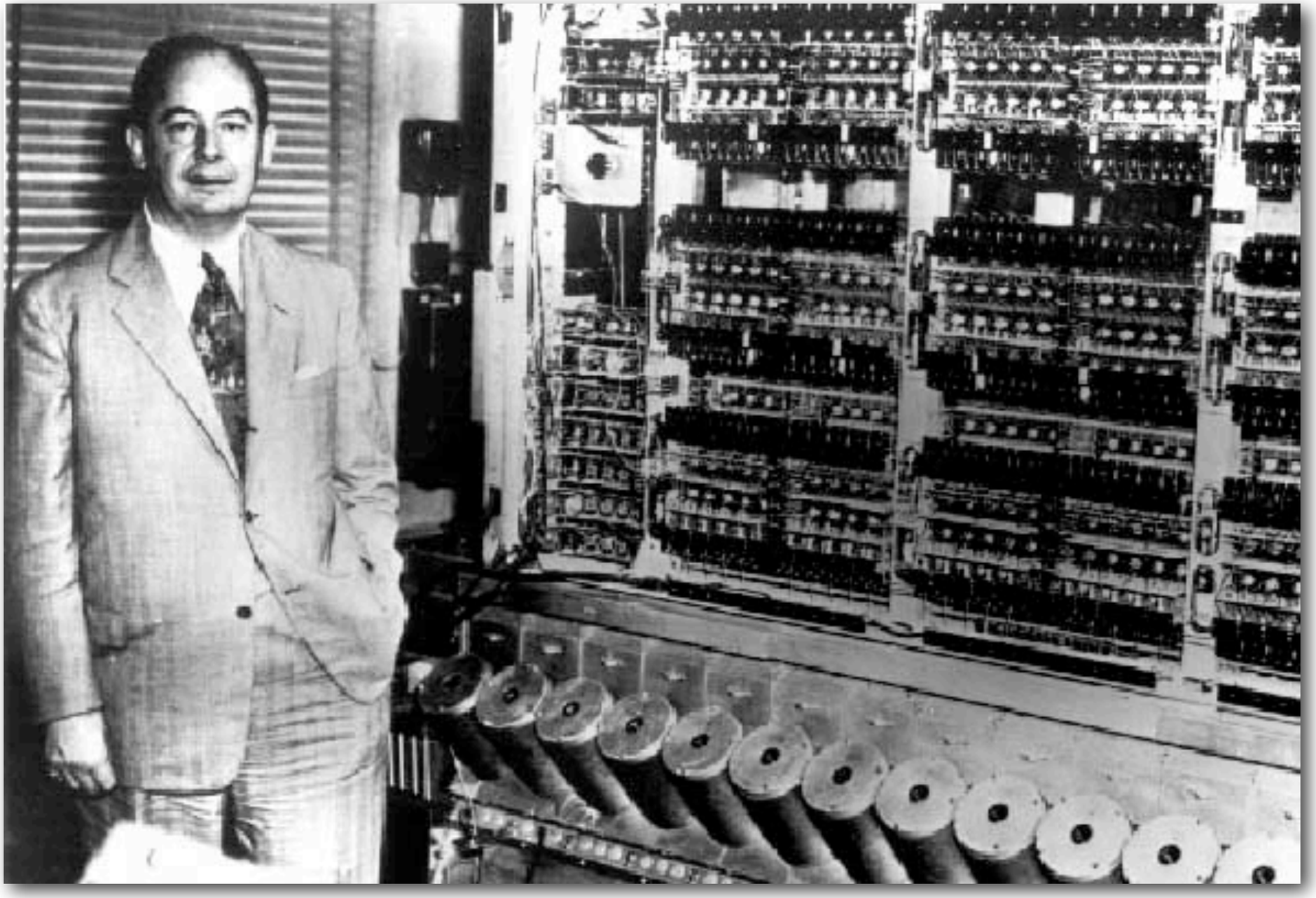
- Kip R. Irvine: *Assembly Language for x86 Processors*, 6th Edition, Prentice Hall, February 2010



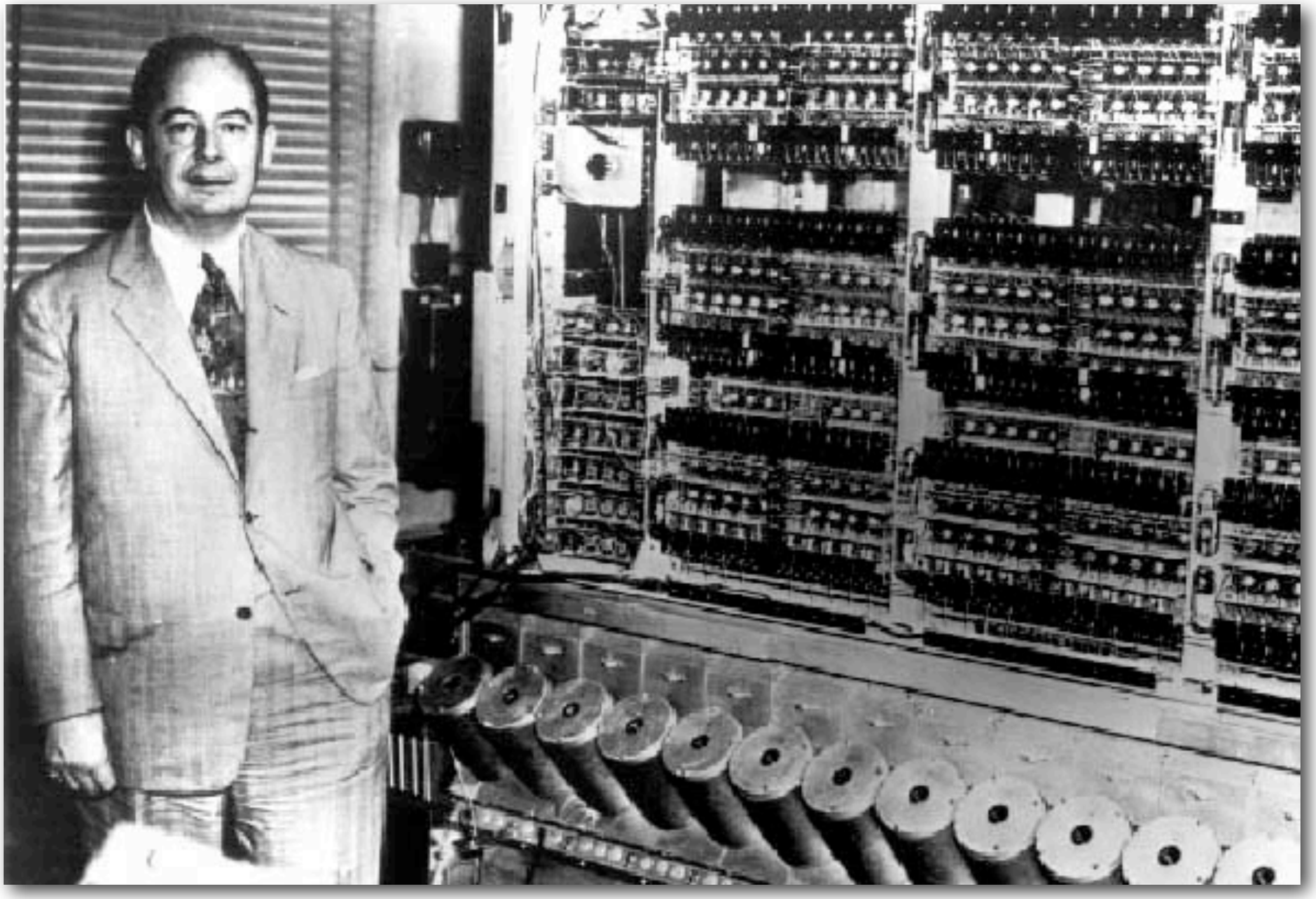
Hardware Basics

Processor

- Processor (Central Processing Unit) is the "heart"
- Main source of all activities
  - (Almost) all other components are passive and are controlled by the processor
- Different components
  - Arithmetic logical unit (ALU)
  - *Registers*: set of specific memory cells
  - *Control unit* for instructions / co-ordination

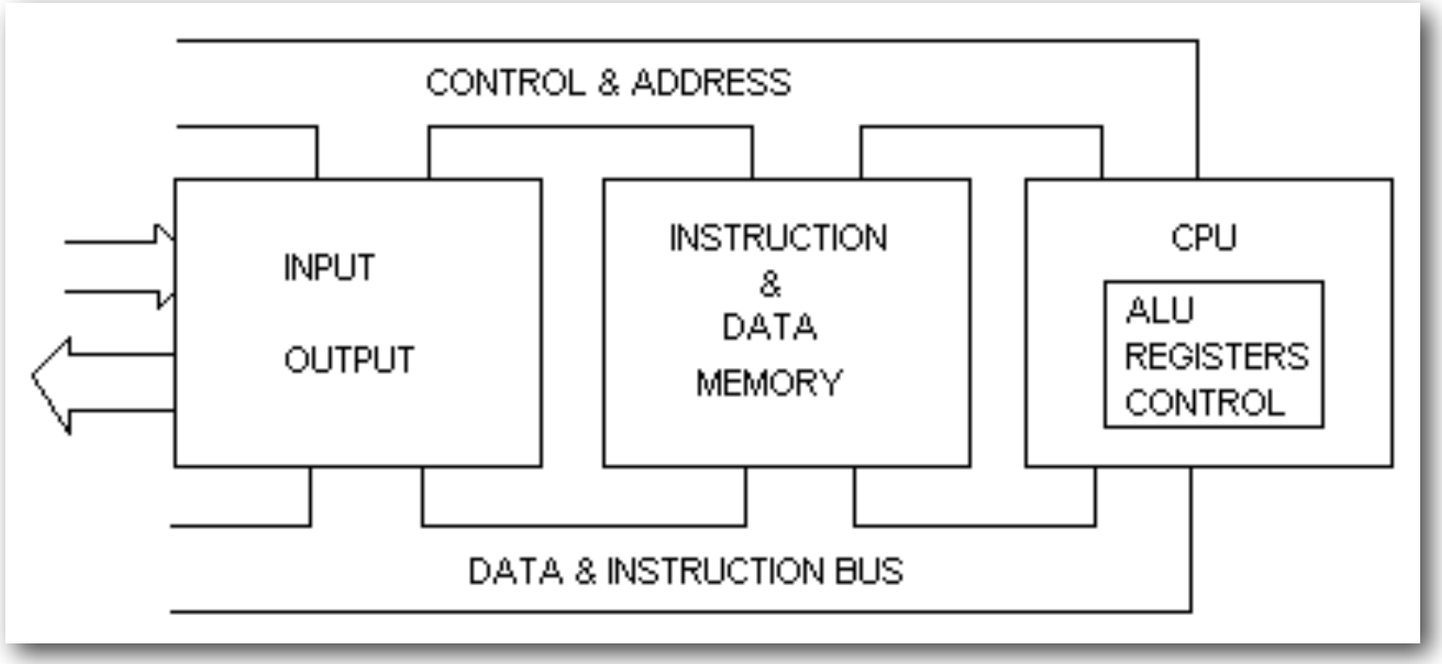


Von Neumann Architecture



Von Neumann Architecture

- Components
  - CPU
  - Memory
  - I/O devices



- Bus is used to address memory units
- Bus is central mechanism for communication between different components

Source: http://www.compeng.dit.ie/staff/tscarff/architecture/neumann_harvard.gif

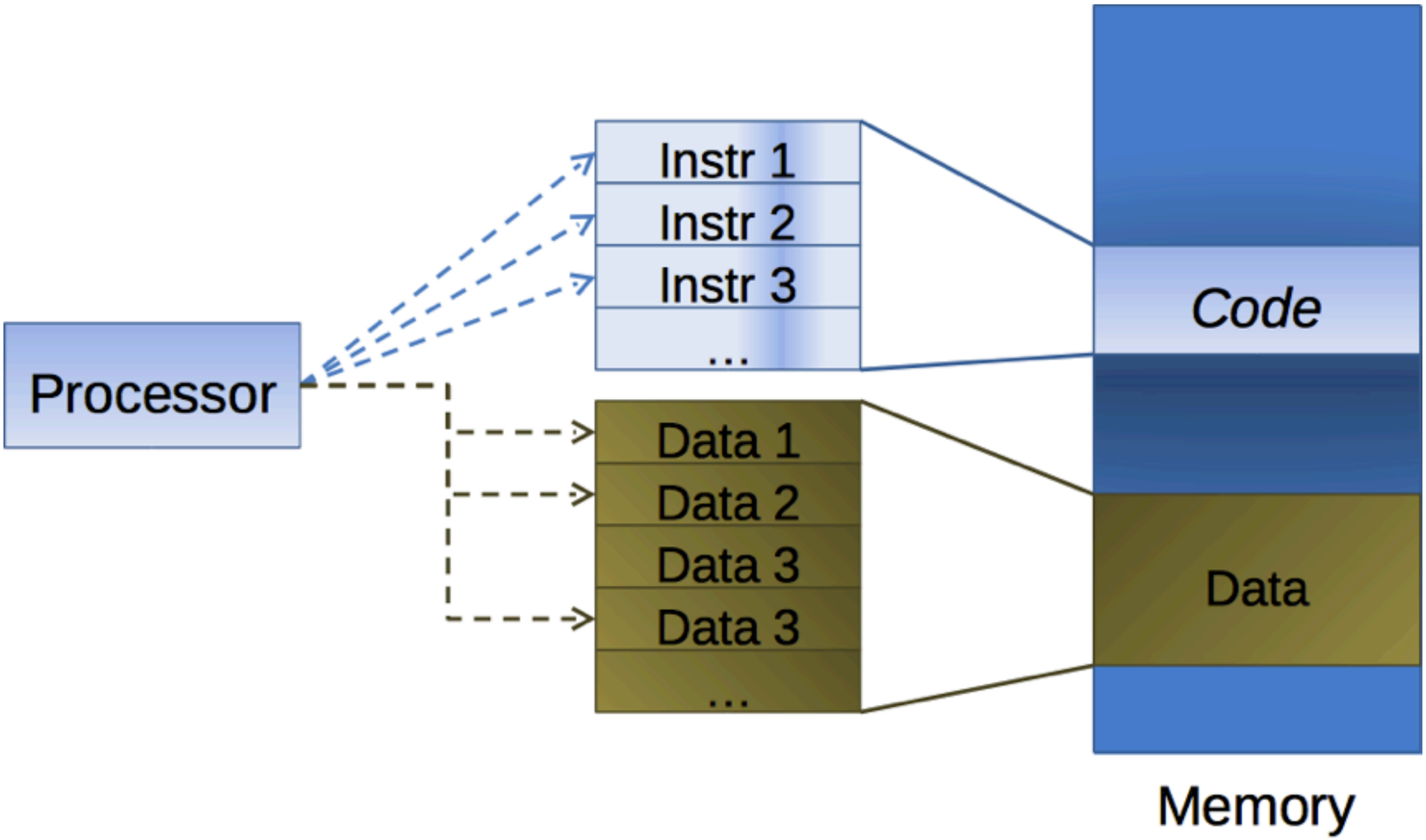
Von Neumann Architecture

- *Data bus* transfers data to read/write
- *Address bus* transfers access addresses
- *Control bus* co-ordinates access to bus and assures correctness of transferred data
- Width of address bus determines memory size (2^{32} byte conforms to 4GB)
- Width of data bus determines number of cycles for data transfer

Von Neumann Architecture

- Program is *sequential* collection of instructions
- Instructions are stored together with data in memory (*Von Neumann Principle*)
  - Can lead to many problems (e.g., buffer overflows)
- Processor reads next instruction from memory and executes the specified action with the corresponding data (*Von Neumann Cycle*)
- Memory is *Random Access Memory (RAM)*

Von Neumann Architecture

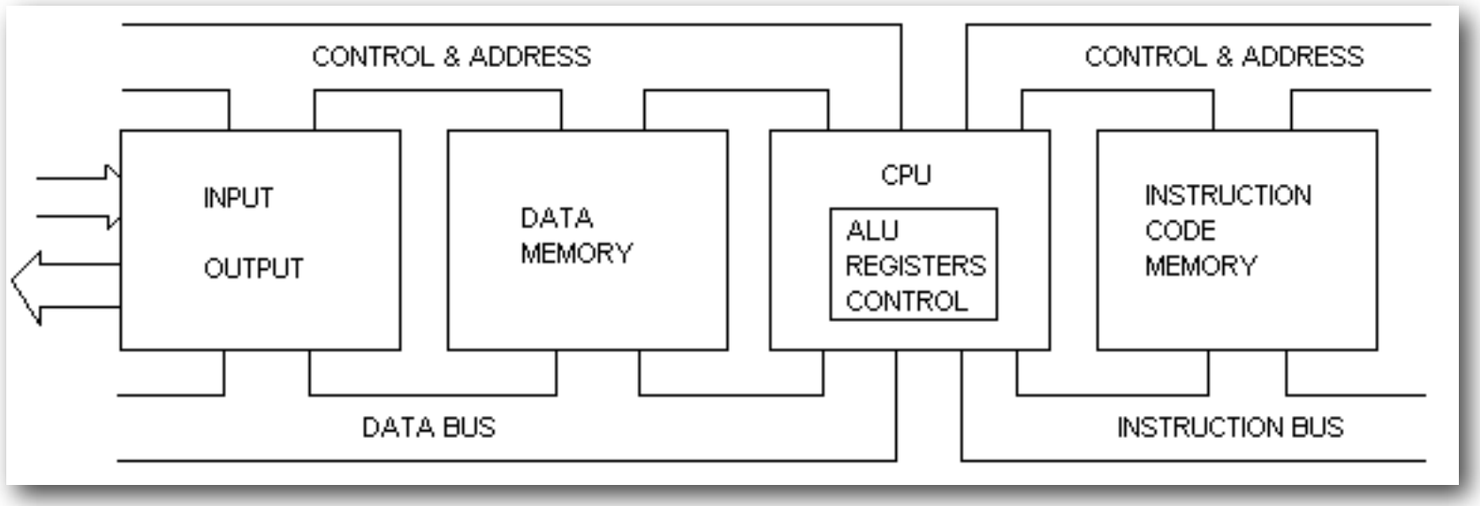


Von Neumann Cycle

- Always the same cycle
  - Fetch
  - Decode
  - Fetch Operands
  - Execute
- Each architecture has its own instructions
- Set of all instructions is called *instruction set architecture* (ISA)

Harvard Architecture

- Another CPU design, not based on Von Neumann Principle
  - Main difference is that there is a separation between instructions and data



  - First implemented in Mark I, now Atmel AVR, PICs
  - Not covered in detail in this class

Source: http://www.compeng.dit.ie/staff/tscarff/architecture/neumann_harvard.gif

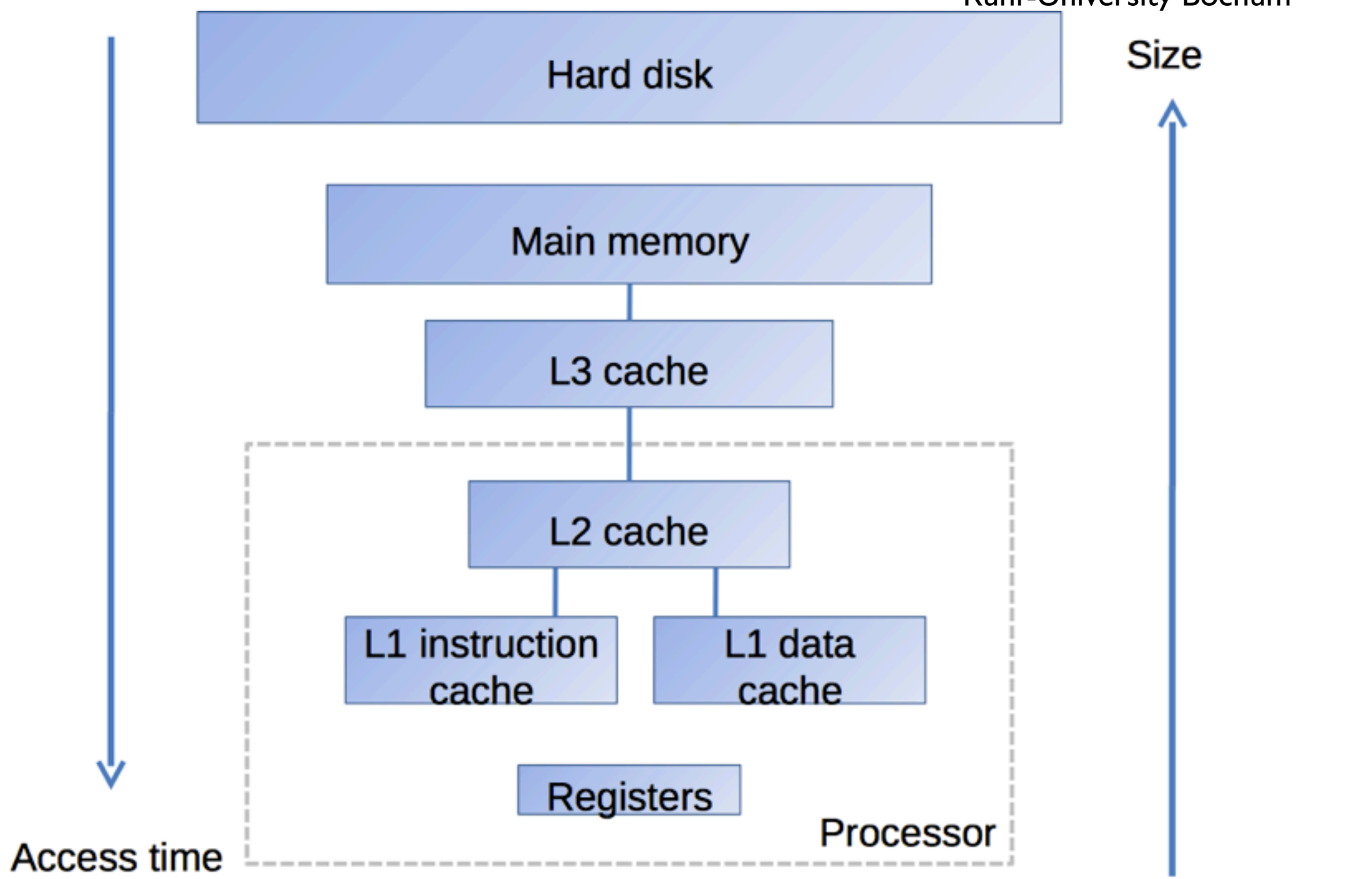
Modified Harvard Architecture

- Hybrids of Harvard and von Neumann architecture
  - Most modern processors have a CPU cache which partitions instruction and data
  - Lots of parallelization within the CPU
    - Almost no pure Harvard/Von Neumann machines
- Examples of modified Harvard architecture:
  - x86 processors
  - ARM cores, MIPS, PowerPC, ...

Registers

- Memory access available on (almost) all CPU architectures is based on *registers*
- Main reason: performance
  - Registers are fast, local memory units
  - Near the CPU, very fast hardware
- Can be accessed similar to variables
  - See exercise on this topic

Memory Hierarchy



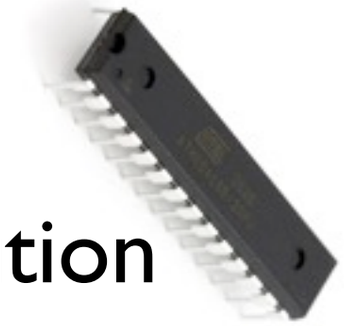
Registers

- Load value from memory address `X` to register `R0`
- Add register `R0` and `R1`, store result in `R2`
- Multiply `R2` by `1234`
- Push `R2` onto the stack
- Call procedure at address `Y`
- Write value from register `R0` to memory address `Z`
- ...

RISC vs. CISC

- Historically, there are two concurrent approaches to design CPUs: *RISC* and *CISC*
- CISC (*Complex Instruction Set Computer*)
  - Only a few *general*, many *special purpose* registers
  - Many different and complex instructions (e.g., loop instructions), encode high-level concepts
  - No common length for instructions
  - Typical example: Intel x86

RISC vs. CISC



- RISC (*Reduced Instruction Set Computer*)
  - Simpler instructions might lead to faster execution
  - Many general purpose registers
  - Only necessary instructions, rather low-level
  - Sometimes fixed instruction length (e.g., all instructions are 32 bit long)
  - Examples: ARM, SPARC, PowerPC, MIPS

- Sometimes hard to have a clear cut / hybrid concepts

Example

strlen

CISC (x86)	RISC (ARM)

`mov edi, str_ptr`
`mov ecx, -1`
`mov al, 0`
`cld`
`repne scasb`
	`ldr r0, str_ptr`
	`ldr r1, 0`
	`loop:`
	`ldr r2, [r0+r1]`
	`cmp r2, 0`
	`jz end`
	`add r1, r1, 1`
	`jmp loop`
	`end:`

RISC vs. CISC

- CISC instructions are typically easier to understand for humans
  - A bit similar to higher-level programming logic
  - Since there are many different instructions, it is typically easier to “see” the intention behind a code snippet
- RISC is more similar to the actual physical construction of a machine
- Distinction between RISC/CISC has blurred in practice

Load-Store Architectures

- Memory access only via dedicated load and store instructions
- All other instructions *only* operate on registers
- Optimizes memory access and increases performance
- Memory contents are loaded earlier, better *pipeling* possible

Load-Store Architectures

Example: ARM (Load-Store)

Valid	Invalid
`ldr r0, [pointer]` `add r1, r1, r0`	`add r1, r1, [pointer]`

Load-Store Architectures

Example: ARM (Load-Store)

Valid	Invalid
`ldr r0, [pointer]` `add r1, r1, r0`	`add r1, r1, [pointer]`

Example: Intel (No Load-Store)

Valid	Also valid
`mov ebx, [pointer]` `add eax, ebx`	`add eax, [pointer]`

Intel x86 Architecture

Intel x86 (aka IA-32)

- Common CPU architecture: *Intel 32bit x86* (also *IA-32*)
  - Based on 8086 (1978)
  - Success due to IBM PC
  - Continuos development (286, 386, Pentium, ...)
  - 32bit "basic architecture" exists since 386
- 64bit extension AMD64/Intel64 is similar
  - ***Not*** IA-64 aka Itanium - *not covered in this class*

Intel x86 (aka IA-32)

- Several manufacturers build processors compatible to x86 processors (e.g., AMD, Cyrix, etc.)
- Historical milestones:
  - 8086: first x86 processor (16bit)
  - 286: *protected mode* is introduced
  - 386: first 32bit architecture + paging
  - 486: integrated *floating point unit* (FPU)

Architecture Specifics

- 32bit system
- CISC CPU
  - Few general purpose registers, but rather many special purpose registers
  - A large number and complex instructions
- Rather expressive and complex memory management

x86 Registers

Important x86 Registers

General Purpose Register

Register	Purpose
`eax`	Accumulator
`ebx`	Base Address
`ecx`	Counter
`edx`	Data
`esi`	Source
`edi`	Destination

Special Purpose Register

Register	Purpose
`esp`	Stack Pointer
`ebp`	Base Pointer
`eip`	Instruction Pointer
`eflags`	Status Flags

Important x86 Registers

General Purpose Register

Register	Purpose
`eax`	Accumulator
`ebx`	Base Address
`ecx`	Counter

Special Purpose Register

Register	Purpose
`esp`	Stack Pointer
`ebp`	Base Pointer
`eip`	Instruction Pointer

This are just the basic registers, many more registers (e.g., FPU, MMX, and SSE) exist, but they are typically not interesting for us.

General Purpose Registers

Register	Typical purpose
`eax`	Accumulator
`ebx`	Base address for addressing
`ecx`	Counter for loops, index
`edx`	I/O data, double-precision operations
`esi`	Memory address for string source
`edi`	Memory address for string destination

General Purpose Registers

Register	Typical purpose
`eax`	Accumulator
`ebx`	Base address for addressing

Note: This is just a convention, compilers and programmers can use the registers arbitrarily. However, several instructions expect the operand in specific registers, for example:

```
loopz _jumpdest
```

Special Purpose Registers

Register	Typical purpose
`esp`	Points to the next free stack element (**decreases**)
`ebp`	Points to current stack frame
`eip`	Points to current instruction
`eflags`	Different flags, typically used for conditional jumps (see next slide)

Note: `eip` and `eflags` registers can only be written and read *indirectly*

Questions?

Contact:
Prof. Thorsten Holz
thorsten.holz@rub.de
@thorstenholz on Twitter

More information:
http://syssec.rub.de
http://moodle.rub.de