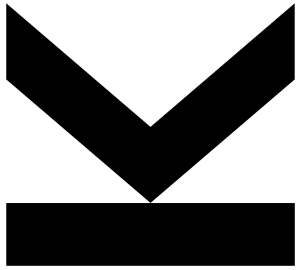


Computer Architecture



Systems Programming



Michael Sonntag
Institute of Networks and Security



Motivation

- Deeper understanding of programming
 - What goes on “inside of the machine”
 - Example: How are functions called (only one return value???)
- Optimization
 - Be careful – This is the **LAST** resort and **ONLY after** profiling!
 - Only if you are an expert – Compilers are **GOOD** at optimizing!
- Embedded Systems
 - Understanding and programming them
 - These are often “raw” machines without any operating system
- Reverse Engineering
 - Investigating malware → no source code!
- Security
 - What are pointers and why are they dangerous (and why can't we live without them, at least on some layer)?

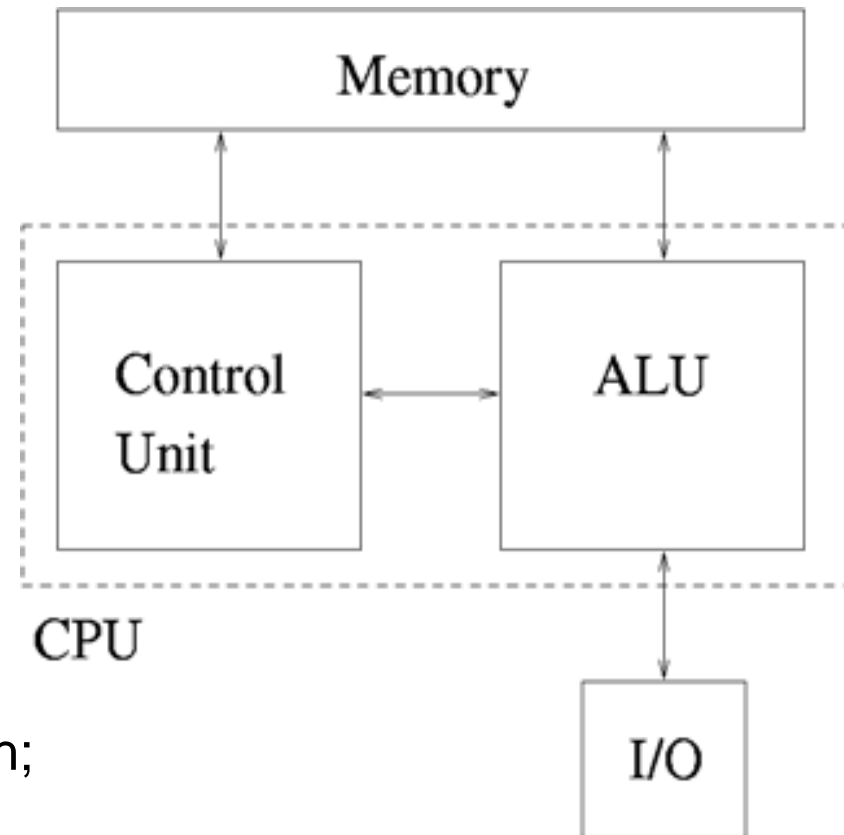
Repetition: Von Neumann architecture

■ Components

- ☐ Arithmetic and Logic Unit (ALU)
 - Includes “Accumulator”
- ☐ Control Unit
- ☐ Memory
- ☐ I/O
- ☐ Bus

■ Data **and** programs stored in same memory

- ☐ Harvard architecture:
Separate memory for program and data
- ☐ Does not change assembly much;
code and data always have
different “instructions”



CPU (simplified!)

■ CPU (Central Processing Unit)

- ☐ Instruction fetcher & decoder
- ☐ Data bus
- ☐ Registers
 - General-purpose registers
 - Special-purpose registers: Program counter, status register
- ☐ ALU: Executes arithmetic and logic instructions

■ Fetch-execute cycle

- ☐ Fetch next instruction from memory
- ☐ Decode it
- ☐ Fetch operands
- ☐ Execute instruction
- ☐ Store result

- Note: **Interrupts** are also handled in this cycle but omitted here for clarity (not handled in this course anyway)!

Registers

■ Registers

- ☐ Designed to hold exactly one “word”
 - ☐ “Word” = 4 bytes for 32-bit architectures, 64-bit→8 bytes...
- ☐ Small and **very** fast memory in CPU
- ☐ Provides quick access to frequently used values
- ☐ Reduces memory traffic
- ☐ Compilers can avoid multiple evaluation of common subexpressions

■ Registers are sometimes implemented as register files/banks

- ☐ Array of registers in CPU
- ☐ Used e.g. for task switching

■ Basically 2 **types**

- ☐ General Purpose Registers (GPR): Used for everything
- ☐ Special Purpose Registers (SPR)
 - Hold status of CPU or other hardware
 - Can only be used for one/some limited uses, sometimes read-only

History of x86 (Intel) processors (1)

■ 8086

- 1978, **29.000** transistors, one of the first single-chip 16-bit microprocessors; 4,77 MHz clock speed
 - Can do everything the latest ones can do too (Turing complete), just **much** more slowly!

■ 80286

- 1982, 134 K transistors, original platform for MS Windows

■ i386

- 1985, 275 K transistors, expanded architecture to 32 bits, first machine that could support a UNIX operating system

■ i486

- 1989, 1,9 M transistors, integrated floating point unit onto chip

■ Pentium

- 1993, 3.1 M transistors, minor extensions to instruction set

History of x86 (Intel) processors (2)

■ Pentium/MMX

- 1997, 4.5 M transistors, added MMX instructions to instruction set

■ Pentium II

- 1997, 7 M transistors, merged PentiumPro and Pentium/MMX

■ Pentium III

- 1999, 8.2 M transistors, added another class of vector instructions, data can be packed into vectors of 128 bits

■ Pentium 4

- 2001, 42 M transistors, added 8-byte integer and floating-point formats to the vector instructions, along with 144 new instructions for these formats

■ ...

■ AMD Zen2 Epyc Rome

- 39,54 billion transistors (8 dies, 64 cores)

Comparison:

128Gb memory chip: 137,438,953,472 transistors
16 chips on a single 256GB DDR4 RAM module

Instruction Set Architecture (ISA)

- Instruction Set: Set of machine instructions for a specific processor
- RISC vs. CISC
 - Reduced Instruction Set Computing (RISC)
 - Small set of simple instructions
 - Easy to decode
 - Large code size
 - ARM, MIPS, SPARC
 - Complex Instruction Set Computing (CISC)
 - Large set of complex and powerful instructions
 - More complex decoding than RISC
 - Small code size
 - IA-32; See also: IA-64 (=Itanium!) / Intel 64 (=IA-32 extended)
- Instruction Set Architecture
 - Part of the computer architecture related to programming
 - Registers, instruction set, addressing modes, memory architecture...

Data access methods (1)

- Different addressing modes (most important & used in x86)
 - ☐ Register: Use CPU register
 - ☐ Immediate: Provide value directly as number
 - ☐ Direct: Specify memory address directly as number
 - ☐ Indirect: Take address from a register, perhaps with additions
 - Several registers, multiplication factors, add static numbers, ...
 - ☐ Inherent: Part of the instruction
 - ☐ IP-based: Relative to instruction pointer
- Register
 - ☐ Instruction specifies register to access data
 - ☐ Example: copy data from **some register** to somewhere
- Immediate
 - ☐ Data is embedded in the instruction (=within the machine code)
 - ☐ Example: initialize a register to **seventeen**

Data access methods (2)

■ Direct

- ☐ Instruction contains memory address to access
- ☐ Example: load the data at [address 4000](#)

■ IP-based (IP = Instruction Pointer)

- ☐ Not really data access, as typically used for control structures
 - E.g. jumps relative to current position (“10 bytes forward in program”)

■ Inherent

- ☐ The memory address/register is defined by the CPU builder and is [“part” of the instruction](#) itself
- ☐ Rare in absolute form (memory); typically used for flags
 - Example: CLC/CLI/CLD = Clear carry/interrupt/direction flag
 - =3 different instructions, not “clear” + “which flag”!
 - Actually, encoding is “clear flag” + “bits determining which flag”...
 - X86 opcodes: F8/FA/FC (CMC=F5...); STC/STI/STD=F9/FB/FD
- ☐ Very common for specifying parts of complex instructions
 - E.g. division → Destination register(s) are hardcoded (=unchangeable)

Inherent example: DIV

- Typical form: DIVQ r/m64

- Meaning: Divide something by either a register or a 64-Bit memory content; assuming it is an unsigned value

- Note: No immediate possible!

- Result:

- RDX:RAX is what we divide (128 Bit for a 64 Bit division!)

- RDX is the “higher” part, RAX the “lower” part

- Result: RAX

- Remainder: RDX

- Flags:

- All of CF (Carry), OF (Overflow), SF (Sign), ZF (Zero), AF (Auxiliary Carry), PF (Parity) → Undefined

- Can be the same, can be different, no guarantee for anything!

- Overflow produces an error instead of setting the flag

- Similar: IMUL r/m64, but see also: IMUL r/m, r or IMUL i, r/m, r

Data access methods (3)

■ Indirect

- Instruction specifies register that contains the address to the data
- Example: load data from address; **address** is found in a **register**

■ Indirect-Indexed

- Instruction contains **memory address** and additionally specifies an **index register** to offset address
- Example: load the data at address (4000 + index register)
 - If index register contains 4 → we load data from address 4004
- On x86 processors a multiplier for the index register is possible
 - Limited: Only 1, 2, 4, and 8 are allowed as multipliers

■ Indirect-Base Pointer

- Contains **memory address + index register + additional offset**
 - Adds a displacement before lookup
- Typically used for local variables or records
- Index register can again have a 1/2/4/8 multiplier

Instruction descriptions

- Descriptions of instructions contain abbreviations
 - R, R/M, I...
 - These describe the possible operands of the instruction
 - “What can you put here”
- Possibilities:
 - I = Immediate → A number embedded in to the machine code
 - Maximum is 32 bit → even on IA64 never a 64 bit value!
 - R = Register → Typically any register, but restrictions are possible
 - M = Memory → Any memory address (so also 64 bits possible!)
 - Typically also involves a segment selector, but not used here
 - In 64 bit mode these are (mostly) treated as always zero
 - I/O Port → Communication with “peripherals”, like Interrupt controller, serial port etc. Can be immediate value or DX register on IA64

Instruction descriptions

- Cheatsheet example: `mov src, dest`
 - `R, R/M` → Move from register to register
Move from register to memory
 - `R/M, R` → Move from register to register
Move from memory to register
 - `I, R/M` → Move immediate to register
Move immediate to memory
- What is NOT possible:
 - Move from memory to memory (`M, M`)
 - Limitation of hardware
 - Move from register to immediate (`M, I`)
 - Does not make sense
 - `mov array(data(rax,rdi,2),rsi,8),rbx`
 - Only a single “base+(offset+factor*scale)” is possible

X86-64

Memory models

- Memory is not address directly (=physically), but through the MMU
- Supports 3 different **memory models**
 - Flat
 - Memory appears as single contiguous address space (=linear)
 - Segmented
 - Memory appears as a group of independent address spaces (segments)
 - Linear address \approx Segment selector + offset
 - Also used for security purposes
 - CS = Code Segment \rightarrow Register content cannot be changed by application (only OS), memory content can only be executed
 - Real mode
 - Memory model of the 8086 processor; Intel CPUs boot into this
 - Provided for compatibility; special version of segmented mode
 - Limited to 2^{20} Bytes linear address space
- Logical memory: 64 Bit linear address space
 - Physical memory support (currently) limited to 46 Bit

Registers

■ 16 64-bit **General Purpose Registers**

- RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, and R8-R15

■ 6 16-bit **Segment Registers**

- CS, DS, SS, ES, FS, and GS
 - In 64 Bit CPU mode (≠memory model!) CS, DS, ES, and SS are always treated as 0

■ 32-bit **EFLAGS** (Program Status and Control Register)

- Status of the program being executed
- Group of status, control, and system flags
- 64 Bit mode: RFLAGS; upper 32 Bits are reserved (unused)

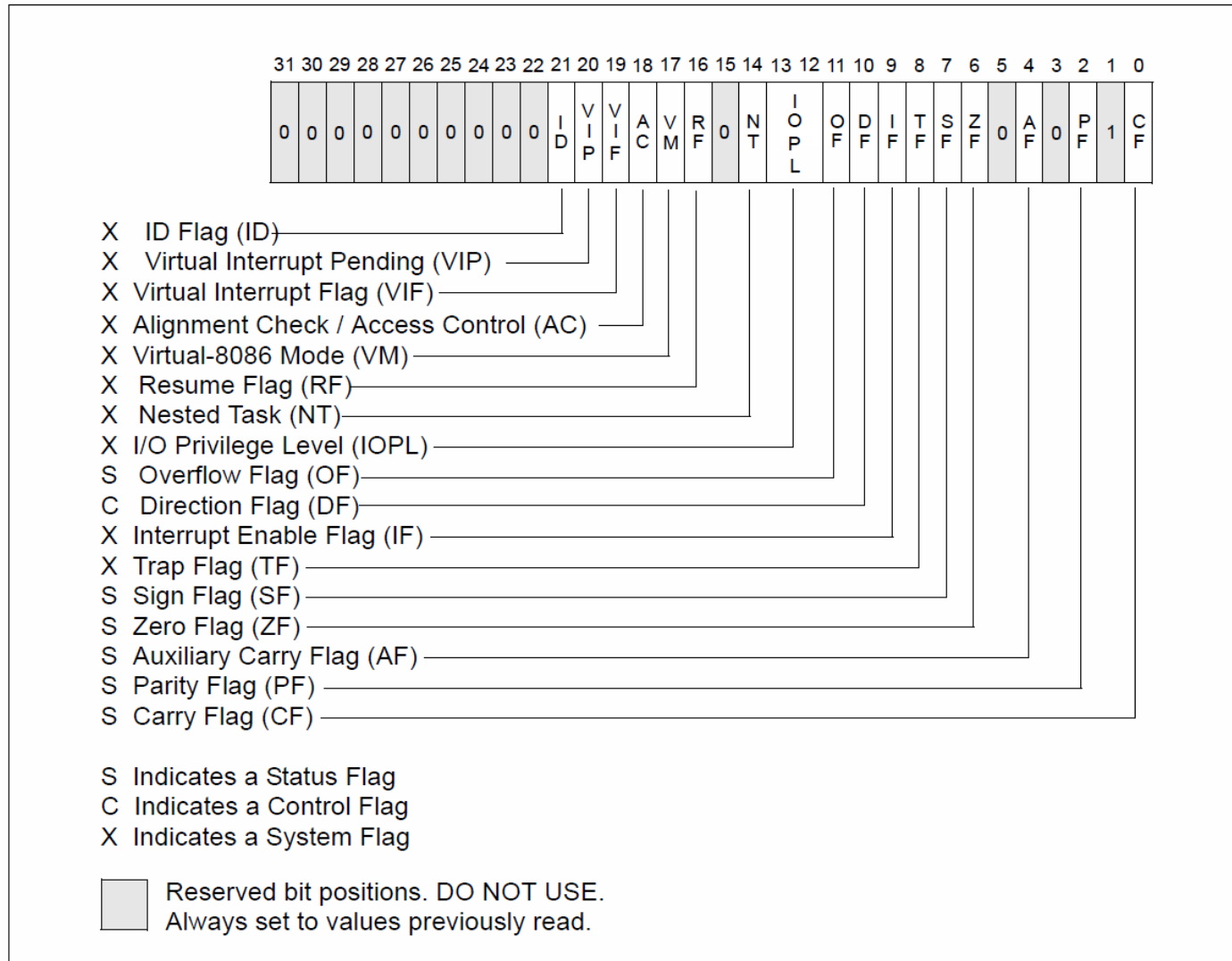
■ **RIP** (instruction pointer) register

- Contains address of the next instruction to be executed

■ Plus lots of special registers

- Floating point, MMX, XMMX, Control, Debug, Memory management, Profiling, System information/management...

Flags

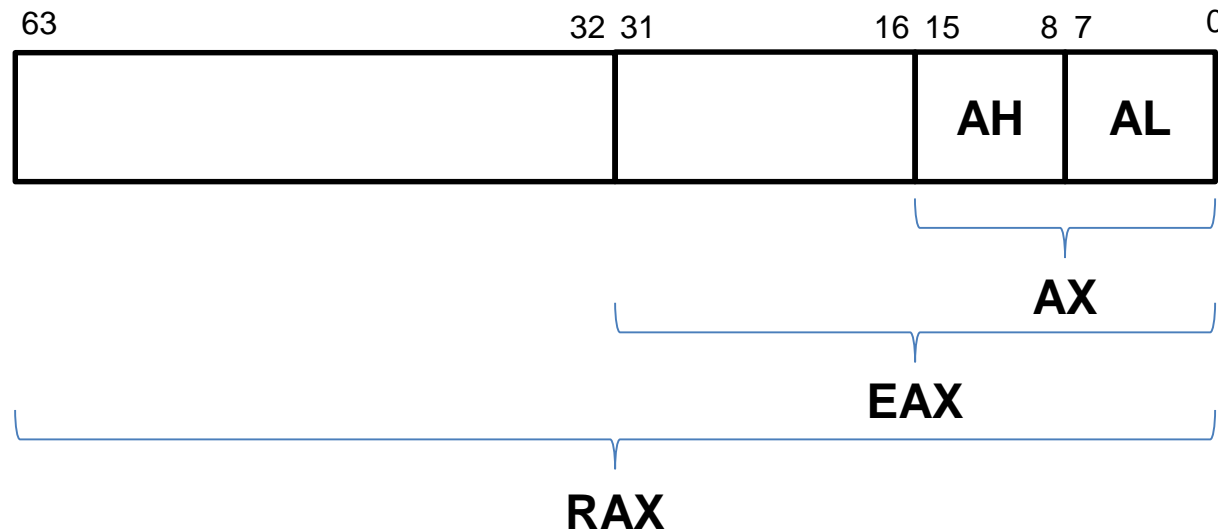


Registers

- RAX: Accumulator (“A register”)
 - Operands, results, return values from functions...
- RBX: Pointer to data in the DS segment
 - Today: General register
- RCX: Counter for strings and loops instructions
 - Today: Mostly a general register
- RDX: I/O pointer
 - Today: General register
- RSI: Data in DS; Source index (Today: Mostly a general register)
- RDI: Data in DS; Destination index (Today: Mostly a general register)
- RSP: Stack pointer
- RBP: Pointer to data on stack (“Base pointer”); sometimes gen. reg.
- R8-R15: General registers

Registers

- The layout of registers is complex: It allows subpart access
 - Same applies to other registers: EBX, ECX, EDX
 - Limited to RSI, RDI and others (SIL, SI, ESI; no “SIH”; SIL only for some instructions/combinations available)
 - “E..” only available on 32 Bit CPUs/modes
 - “R..” only available on 64 Bit CPUs/modes



- NOTE: This is **one 64** Bit register – only subparts with **different names!**

Register naming

Register	8 Bit (Byte)	16 Bit (Word)	32 Bit (Doubleword)	64 Bit (Quadword)
Accumulator	AL / AH	AX	EAX	RAX
Data pointer	BL / BH	BX	EBX	RBX
Counter	CL / CH	CX	ECX	RCX
I/O pointer	DL / DH	DX	EDX	RDX
Source index	SIL / -	SI	ESI	RSI
Dest. Index	DIL / -	DI	EDI	RDI
Base pointer	BPL / -	BP	EBP	RBP
Stack pointer	SPL / -	SP	ESP	RSP
R8-R15	R8B-R15B / -	R8W-R15W	R8D-R15D	R8-R15

64 Bit mode only!

Register peculiarities in 64 Bit mode

- When calculating, the result size depends on the operands
 - ☐ 64 Bit → 64 Bit result
 - ☐ 32 Bit operand → 32 Bit result, **upper 32 Bits are zeroed**
 - ☐ 16 Bit operand → 16 Bit result, upper 48 Bits remain **unmodified**
 - ☐ 8 Bit operand → 8 Bit result, upper 56 Bits remain **unmodified**
- Limitations on accessing byte registers
 - ☐ Legacy **high** bytes (AH-DH) cannot be in the same instruction as new single-byte registers (SIL, DIL, R8B etc)
 - ☐ Legacy **low** byte registers are not restricted!
 - ☐ AL + SIL → OK; AH + SIL → Not possible
- Immediate values (number is directly encoded into the instruction) are still limited to 32 Bit

Basic assembly instructions (1)

■ Data Transfer Instructions

- ☐ MOV: Move data (actually: copy data)
- ☐ XCHG, BSWAP: Exchange data, Endian conversion (byte swap)
- ☐ PUSH, POP: Manipulate stack

■ Binary Arithmetic Instructions

- ☐ ADD, ADC, SUB, MUL, DIV, IMUL, IDIV
 - IMUL and IDIV are signed and MUL and DIV unsigned operations
- ☐ INC, DEC: Increment, Decrement
- ☐ NEG: Negate (change mathematical sign of number)
- ☐ CMP: Compare data

■ Logical (Bit) Instructions

- ☐ AND, OR, XOR, NOT: Bit operations
- ☐ ANDN (Not A and B),

Basic assembly instructions (2)

■ Shift and Rotate Instructions

- SAR, SHR, SHL, ROR, ROL, RCR, RCL

- Shift Arithmetic/Logical Right, Shift Logical Left, Rotate Right/Left, Rotate through carry

■ Bit and Byte Instructions

- BT, BTS, BTR, BTC, BSF, BSR

- Bit Test, Bit Test Set, Bit Test Reset, Bit Test and Complement, Bit Scan Forward/Rev.

■ Flag configuration

- CLC, CLD, STC, STD: Clear carry/direction, Set carry/direction

■ Miscellaneous

- LEA: Load Effective Address
- NOP: No Operation

Basic assembly instructions (3)

■ Control Transfer Instructions

- ☐ JMP, JE/JZ (ZF), JNE (ZF), JG (SF), JA=JNBE (CF), JAE (CF+ZF), JL (SF), JB (CF), JBE (CF+ZF); JNG/JNA/JNB/...; LOOP
 - Jump, Jump Equal/Zero, Jump Not Equal, Jump if Greater...; Loop with ECX counter
- ☐ CALL, RET
 - Low-Level Procedure support
- ☐ ENTER, LEAVE
 - High-Level Procedure support (calling procedures in C or Pascal; reserves space for local variables and supports easy access to local variable in “higher” procedures)
- ☐ SYSCALL/SYSRET: Fast system call to OS
- ☐ INT: Call interrupt procedure
 - Access to OS in 32 Bit mode on Linux (instead of SYSCALL)

THANK YOU FOR YOUR ATTENTION!

Slides by: Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)