CPU Internals and the x86 Instruction Set

ft. graphics stolen from the internet

Tristan Miller

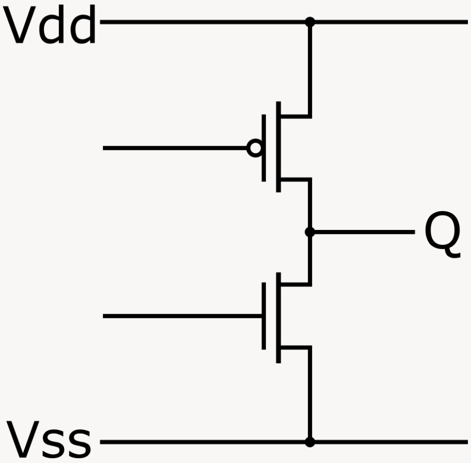
2024

1. making the sand think

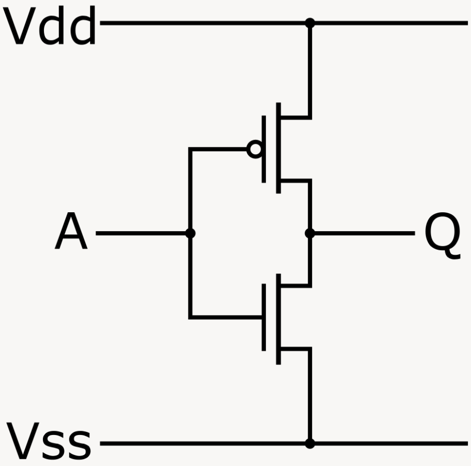
the Transistor

- Invented in 1947 in Bell Labs
- Immensely cheaper, smaller, faster, and less energy-using than vacuum tubes and relays
- Singlehandedly* allowed for the proliferation of computers in everyday life

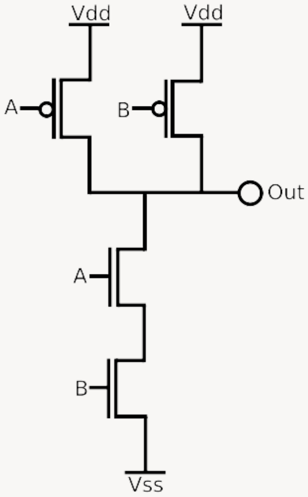
the Transistor

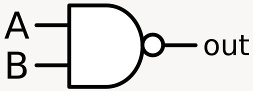


the Transistor



the Transistor





logic gates

- Transistors can be combined to create basic logic operations
    - NOT, AND, OR, NAND, NOR, XOR, XNOR
- More complicated operations can be created as combinations of these simple ones

numbers

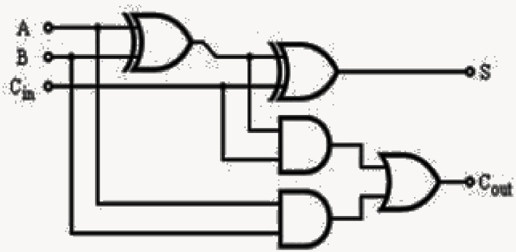
- Wires can be in two* states: on and off
- Computers use binary since it only requires two symbols per place value

numbers (diy slide)

a demonstration of various binary arithmetic operations shall proceed below

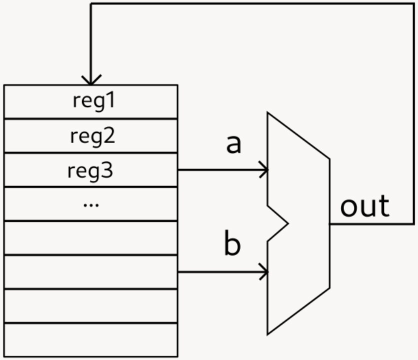
making the computer do numbers

- Full adder: implements a single column of addition
- Chain the C_{out} of one into the C_{in} of the next to add larger numbers



arithmetic AND logic???

- Arithmetic logic unit (ALU) - unit that performs arithmetic and logic
- Where do we get inputs and store outputs? registers!



memory and instructions

- Problem: it doesn't do anything
- Solution: tell it to do things
- Attach some memory containing a list of instructions
- Program counter register stores location of next instruction
- For each instruction:
    1. Fetch it from memory, move the PC to the next instruction
    2. Decode it to determine what to do
    3. Execute it

types of ISAs

- RISC - fairly simple instructions that do one thing each
- CISC - instructions can be more complicated, take more steps to execute

types of ISAs

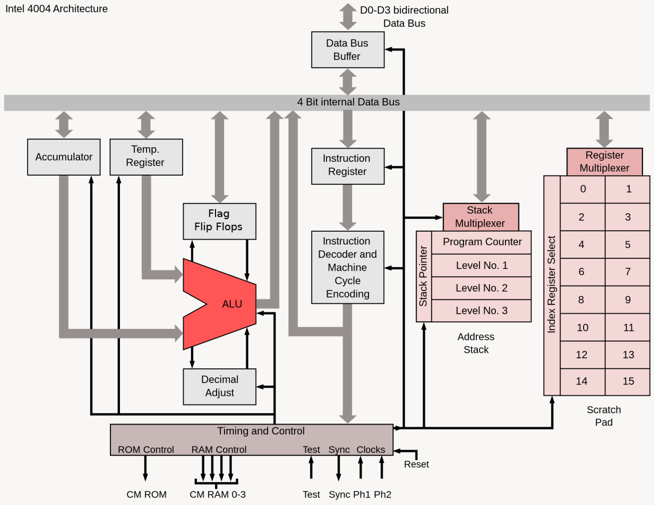
- Load-store: instructions either load/store data between memory and registers or operate on values already in registers
- Register-memory: operations can operate on memory directly

2. how did we get here

1971 - Intel 4004

- First microprocessor
- 4-bit data, 16 registers
- 12-bit addressing
- Separate memory for instructions and data (Harvard architecture)
- Internal function call stack

1971 - Intel 4004



1972 - Intel 8008

- 8-bit data, 7 registers, larger call stack
- 14-bit addressing

1974 - Intel 8080

- 16-bit addressing
- Support for 16-bit operations using pairs of registers
- Stack pointer register replaces internal stack
- Zilog z80 - compatible with Intel 8080, adds additional registers and instructions

1978 - Intel 8086

- Four general-purpose 16-bit registers, 8-bit halves can be accessed as well
    - AX (AL, AH), BX (BL, BH), CX (CL, CH), DX (DL, DH)
- Two index registers (SI, DI), base pointer BP and stack pointer SP
- 20-bit addressing
- wait... so how does that work exactly?

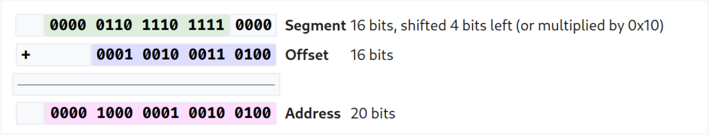
aside - different address and data size

- 8- or 16-bit data is sufficient for many tasks, but memory often needs to be bigger
- 4004 and 8008 used Harvard architecture and had internal call stacks - no issue
- 8086 would need a different way to work around this

aside - different address and data size

- Segment registers - additional 16-bit registers
    - CS (code)
    - DS (data)
    - ES (extra)
    - SS (stack)
- Allow different things (code, data, stack) to be stored in different parts of memory
- Programs that didn't need much memory could set CS=DS=ES=SS
- Programs that needed even more than 64K each for code, data, and stack could change segments during execution

aside - different address and data size



	0000 0110 1110 1111 0000	**Segment**	16 bits, shifted 4 bits left (or multiplied by 0x10)
+	0001 0010 0011 0100	**Offset**	16 bits
	0000 1000 0001 0010 0100	**Address**	20 bits

8086 instruction set

- Similar to previous Intel ISAs
- ALU operations take two arguments - source and destination
    - *`add ax, bx`* (*`ax += bx`*)
    - source can be register, memory, or immediate, destination can be register or memory (at most one argument in memory)
- Many addressing modes, including direct, indirect, and indexed

8086 instruction set

- multiplication and division use ax and dx to store the lower and higher bits of the result
- 256 interrupts
- *`push`*, *`pop`*, *`call`*, and *`ret`* instructions for managing the stack
- string instructions that automatically increment si or di
- floating-point operations via optional 8087 coprocessor

80186 and 80286

- 80186 (1982) - small improvement, adds some new instructions
- 80286 (also 1982) - 16-bit protected mode
    - Virtual addressing expanded memory to 24 address lines (16M)
    - Memory protection prevented programs from accidentally modifying their own code or trying to execute data
    - Privilege levels prevent programs from accessing each other's memory

‘

- 286 protected mode turned out to be kind of terrible, rarely actually used

Aside: the A20 Line

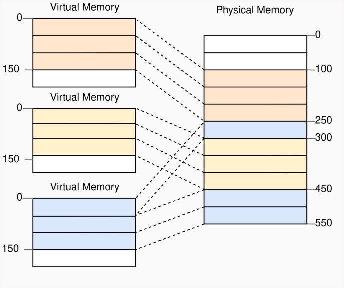
- Imagine: the year is 1984. you are IBM.
- Problem: some programs expected memory to wrap around at the end, increasing address size to 24 bits would break this.
- Solution: disable the extra memory by default!
- Problem: how do we enable it
- Another problem: you are IBM and are incapable of doing things in sensible ways
- Solution: there’s an unused pin on the Intel 8042 keyboard controller!

Intel 80386 (i386)

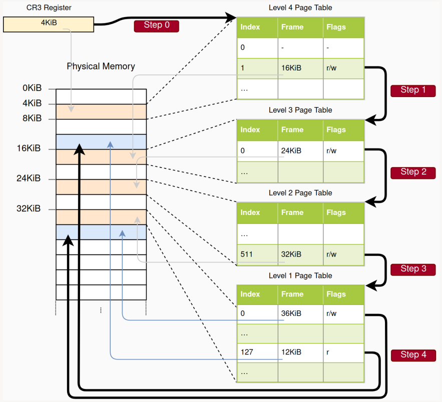
- Released in 1985
- Expands data and registers to 32-bit
    1. EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI
- AX, AL, AH, etc. still work to access lower half of EAX, etc.
- 32-bit protected mode was actually good this time
- 32-bit addressing allowed for 4GB of memory

Paging and memory protection

- Four protection rings, only 0 and 3 used in practice
- Ring 0 code can set up page tables for individual ring 3 processes, each process can only read/write its own memory



Paging and memory protection



Level 4 Page Table

Index	Frame	Flags
0	-	-
1	16KiB	r/w
…		

Level 3 Page Table

Index	Frame	Flags
0	24KiB	r/w
…		

Level 2 Page Table

Index	Frame	Flags
…		
511	32KiB	r/w

Level 1 Page Table

Index	Frame	Flags
0	36KiB	r/w
…		
127	12KiB	r
…		

i486 - i686

- i486 (1989) - integrated FPU
- i586/Pentium (1993) - basic SIMD instructions
- i686/Pentium II (1995)
- Various performance improvements with each new processor - better pipelining, branch prediction, speculative execution, etc

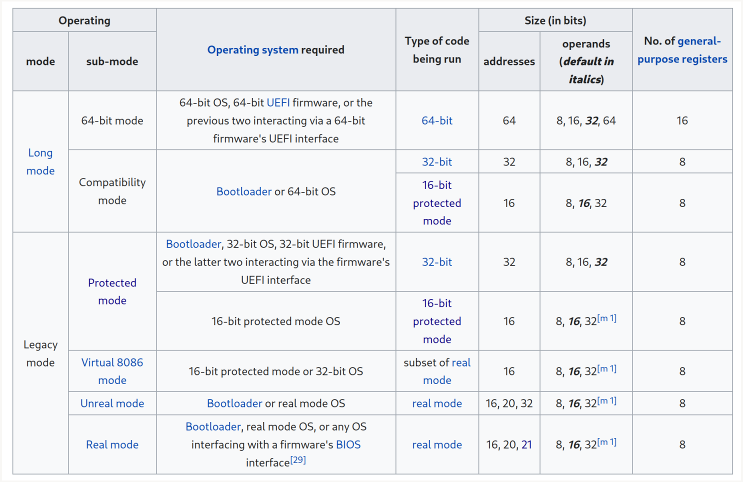
more bits

- Problem: 32 bits (or 4G of memory) is not very much
- Solution: more bits
- 2001: Intel released IA-64 (Itanium) architecture, *NOT* backwards-compatible with x86
- 2003: AMD released AMD64 architecture, which *WAS* backwards-compatible with x86
    - which do you think caught on?

AMD64 / x86_64

- Current most popular architecture for personal computers and servers[citation needed]
- Adds a 64-bit long mode with 64-bit data and addresses*
- Extends general-purpose registers to 64 bits
    - RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI
    - EAX, AX, AH, AL, etc. remain in the lower half of RAX, etc.
- Adds 8 new 64-bit registers: R8 - R15
- Removes* segment registers and other vestigial features
- Additional SIMD extensions

AMD64 / x86_64



Operating		Operating system required	Type of code being run	Size (in bits)		No. of general-purpose registers
mode	sub-mode			addresses	operands (***default in italics***)	
Long mode	64-bit mode	64-bit OS, 64-bit UEFI firmware, or the previous two interacting via a 64-bit firmware's UEFI interface	64-bit	64	8, 16, ***32***, 64	16
	Compatibility mode	Bootloader or 64-bit OS	32-bit	32	8, 16, ***32***	8
			16-bit protected mode	16	8, ***16***, 32	8
Legacy mode	Protected mode	Bootloader, 32-bit OS, 32-bit UEFI firmware, or the latter two interacting via the firmware's UEFI interface	32-bit	32	8, 16, ***32***	8
		16-bit protected mode OS	16-bit protected mode	16	8, ***16***, 32[m 1]	8
	Virtual 8086 mode	16-bit protected mode or 32-bit OS	subset of real mode	16	8, ***16***, 32[m 1]	8
	Unreal mode	Bootloader or real mode OS	real mode	16, 20, 32	8, ***16***, 32[m 1]	8
	Real mode	Bootloader, real mode OS, or any OS interfacing with a firmware's BIOS interface[29]	real mode	16, 20, 21	8, ***16***, 32[m 1]	8

3. in which i attempt a live assembly programming demo