Gmail

Lukáš Hozda <luk.hozda@gmail.com>

Programování - Patnáctá hodina (+ zbytek čtrnácté)

1 message

Lukáš Hozda <luk.hozda@gmail.com> Mon, Mar 11, 2019 at 2:43 PM

Zdravím všechny,
posílám látku z poslední hodiny a zbytek z předposlední. Z předposlední je to ještě ta sčítačka; z poslední je to odčítačka; krátký popis dalších částí CPU; Harvardská, von Neumannova a modifikovaná Harvardská architektura; něco o registrech, mezipaměti a strojovém kódu.

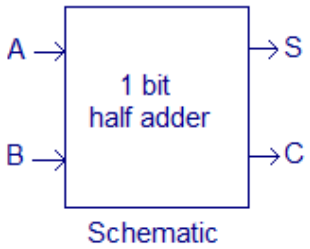
Binární sčítačka (adder) a poloviční sčítačka (half-adder)

Tyto dva okruhy se v ALU používají pro sčítání čísel. Fungují na podobném principu, jako když jste se učili manuálně sčítat na prvním stupni základní školy. Prostě jedete číslici po číslici a když je překročena maximální hodnota číslice (u desítkové soustavy 9 u dvojkové 1), tak si "držíte jedničku" a budete jí přičítat u sčítání další dvojice číslic.

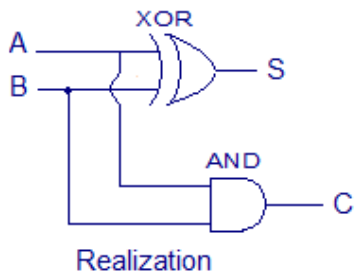
Poloviční sčítačka tedy funguje tak, že sečte dva bity a na výstupu také dva bity vrací. Ten první je součet (anglicky **sum** -> **S**) a ten druhý indikuje, zda si držíme onu jedničku (tomu se anglicky říká **carry** -> **C**).
Zde je obrázek, který obsahuje i pravdivostní tabulku:

Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth table



Schematic



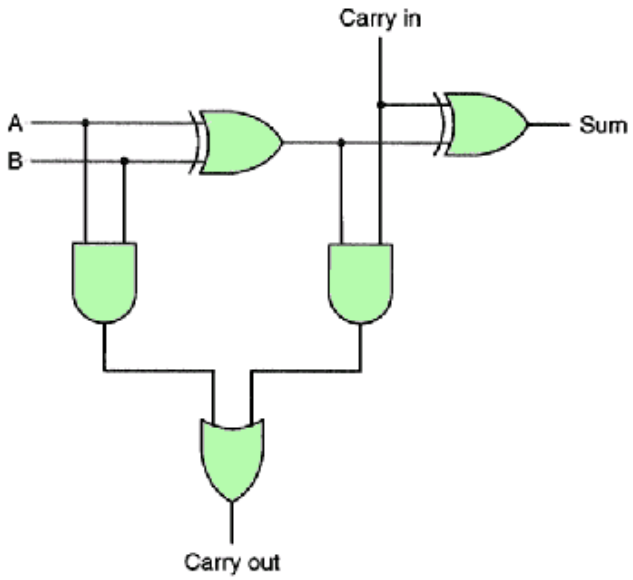
Realization

XOR nám zde udělá faktický součet, jak lze vyčíst i z tabulky. Musíme si totiž uvědomit, že pokud dostaneme na vstup dvě jedničky, tak jejich součet bude nula zbytek jedna. **XOR** vrací jedničku, právě pokud je jen jeden ze vstupů jedna. Proto když dostane 1 a 0, tak na výstupu bude 1, přesně jak to má u sčítání bitů být, a pokud 1 a 1, tak 0.

No a **AND** nám indikuje, že oba vstupy byly jedna a tedy nám vrací ten zbytek 1.

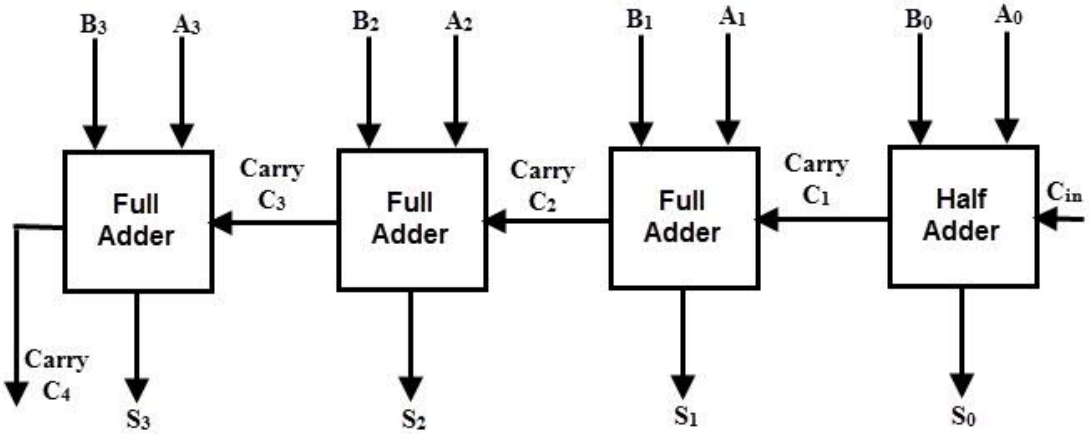
Takhle to ale nestačí, protože musíme být schopni přičíst ještě zbytek. Na to se používá **celá sčítačka**, která je, jak název může vypovídat, složená ze dvou polovičních sčítaček.

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



Na konec je ještě připojený **OR**, který zjišťuje, jestli některá z poloviních sčítaček vrátila **carry**. Zkrátka a dobře, nejdříve sečteme **A** a **B** a pak jejich výsledek sečteme s **C**.

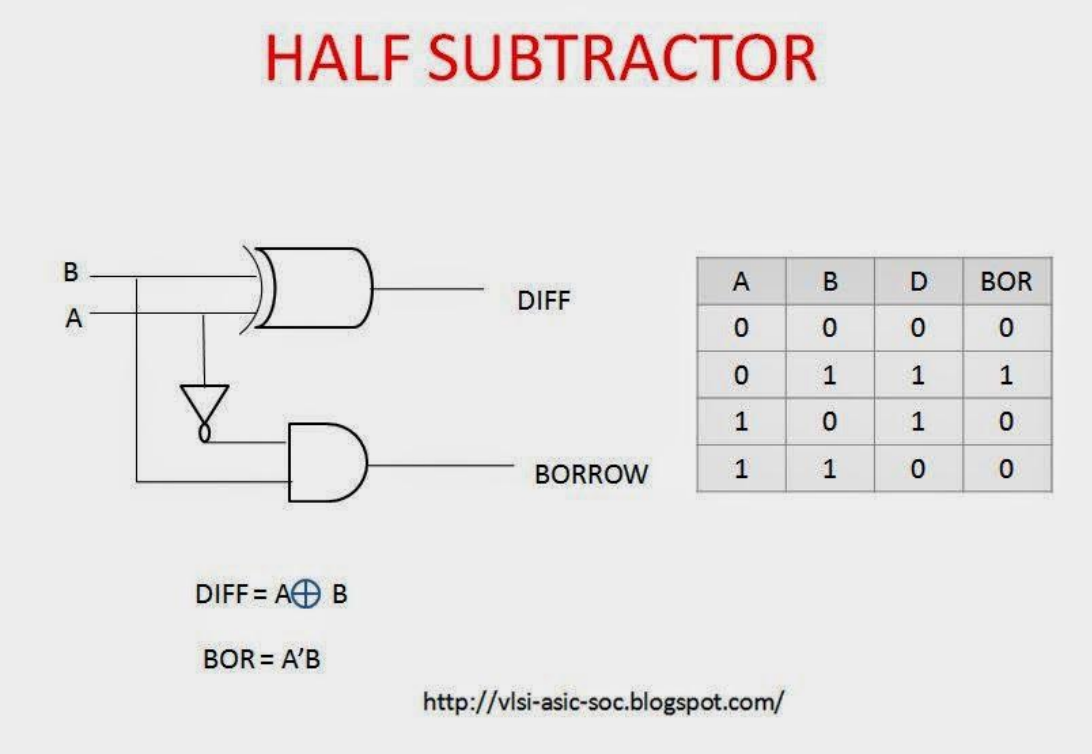
S těmito komponentami už lze postavit opravdovou sčítačku. Zde je schéma čtyřbitové sčítačky. Povšimněte si, že na začátku nám stačí half-adder, protože nemůžeme dostat žádný zbytek jako vstup při sčítání prvních dvou bitů (bity počítáme zprava, mimochodem):



Jak je vidět, jeden zbytek nám takříkajíc "visí". Můžeme ho použít, abychom poznali, že došlo k přetečení (tj. součet čísel je větší než 2^n - 1, kde n je délka čísla v bitech), ale často je ignorován, zkrátka dojde k useknutí.

Binární odčítačka (subtractor) a poloviční odčítačka (half-subtractor)

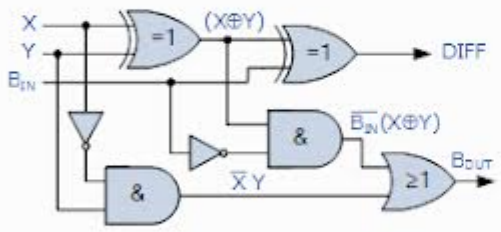
U odčítání to funguje velmi podobně. Zase zde máme poloviční odčítačku, která nepočítá se zbytkem, zde se říká, že si půjčujeme jedničku (anglicky **borrow** -> **B**). Akorát si musíme uvědomit, že odčítání není komutativní, tudíž záleží na pořadí operandů. Odčítáme přeci **od něčeho něco**, místo toho, abychom sčítali dvě čísla dohromady.



A	B	D	BOR
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Takto vypadá poloviční odčítačka. Zbytek dostaneme pouze, pokud je A nula a B jedna, proto musíme znegovat A.

Celá odčítačka je zase analogická celé sčítačce. Zkrátka jsou dvě poloviční odčítačky spojené dohromady a od toho ještě odečítáme zbytek/**borrow** z minula.



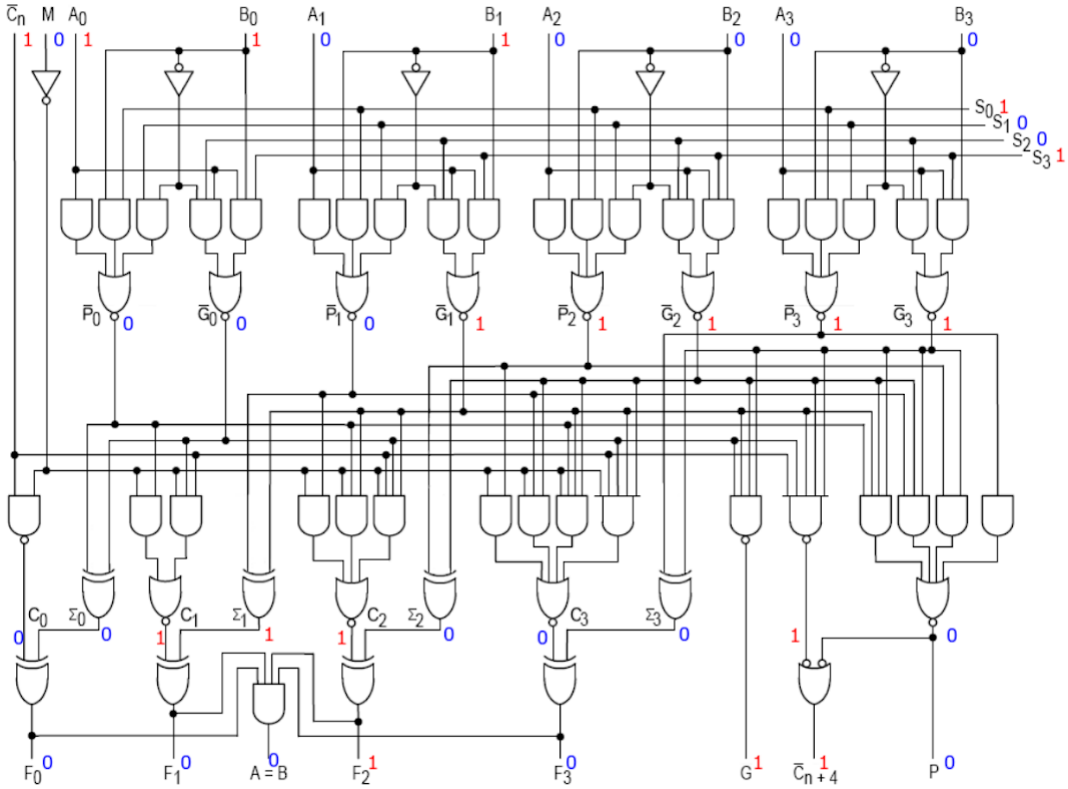
Symbol	Truth Table				
	B-in	Y	X	Diff.	B-out
	0	0	0	0	0
	0	0	1	1	1
	0	1	0	1	1
	0	1	1	0	1
	1	0	0	1	0
	1	0	1	0	0
	1	1	0	0	0
	1	1	1	1	1

Není to žádná velká věda. Povšimněte si, že na logických hradlech je napsána ještě další forma značení (aby v tom nebyl nepořádek, že ano :^)). **XOR** je =1, **OR** je ≥1 a **AND** je &. Zkuste schválně přijít na to proč. My toto značení ale používat nebudeme, už takhle je těch značek dost :D

ALU konkluze

Kromě těchto obvodů se v ALU nachází ještě spousta dalších. Kromě klasických aritmetických a logických operací zde najdete také obvody pro například zjištění, zda je číslo negativní, spočítání nul/jedniček v bitové reprezentaci čísla, bitové posuny a rotace,

Zde je například velmi známe, čtyřbitové ALU 74181:



Operace v něm nejsou úplně klasické. Pochází totiž z doby, kdy elektronika byla ještě poměrně velká a drahá, proto se muselo šetřit a vymýšlet všelijaké vylepšováky aby se obešly technologické limitace. Poskládáním těchto operací však dosáhne procesor kýžené operace.
74181 má ekvivalent (nezapomeňmě na univerzální hradla viz minulý email) pouhých 75 logických hradel, což je na dnešní poměry zanedbatelné množství. Proto se často používá při výuce a je taky vhodné, pokud by si chtěl člověk něco postavit ručně.

Další části procesoru
Procesor má ještě další části, kterými se nebudeme zabývat tak do hloubky. Zde je seznam všech důležitých/univerzálních částí s krátkými popisy:

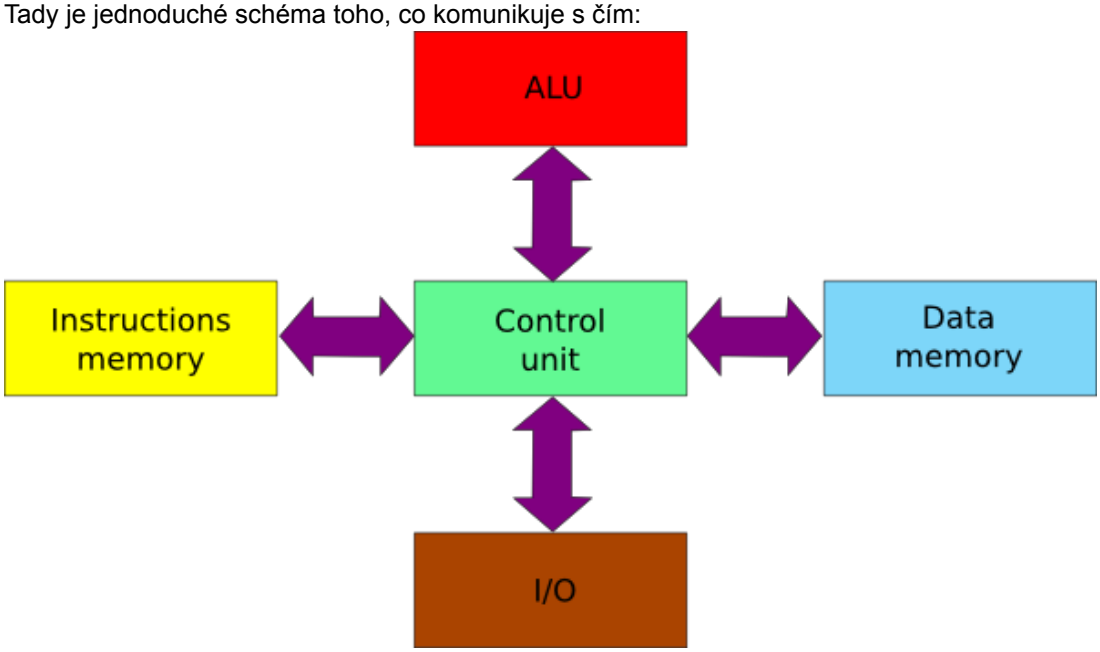
- **ALU** (Arithmetic Logic Unit) - aritmetická a logická jednotka, viz tento a předchozí email
- **CU** (Control Unit) - čte a vykonává instrukce z operační paměti, volá ALU, kdykoliv je to potřeba
- **FPU** (Floating Point Unit) - vykonává operace s čísly s plovoucí desetinnou čárkou. Ty jsou pro hardware poměrně složité a kromě reprezentace čísel
- **Registry** - malé a extrémně rychlé kusy paměti v procesoru. Procesorové instrukce zpravidla operují na datech uložených právě v registrech. Jejich rychlost se měří spíš v procesorových cyklech něž něčem jako Mbps. Většina registrů je přečtena celá za jeden procesorový cyklus.
  - Architektura x86_64, kterou má pravděpodobně váš desktop a laptop, možná i tablet, pokud máte Windows tablet/convertible, má 48 registrů. Někdy se říká, že je to docela málo, což bylo zapříčiněno historickými limitacemi.
  - **16 univerzálních** - zde si může uložit takříkajíc kdo chce, co chce, pokud se to tam vejde
  - **2 statusové**
  - **6 code segmentových**
  - **16 SSE registrů**
  - **8 FPU/MMX registrů**
- **Mezipaměť (cache)** - sem si ukládá procesor větší data a instrukce programů tak, aby byly rychleji dostupné. Cílem je snížit jak přístupovou dobu k datům, tak spotřebu elektřiny. Je velmi blízko procesoru (i když ne tak jako registry), což je to, co jí činí tak rychlou a efektivní, jak jsme si již říkali. (na vzdálenosti od procesoru záleží, čím blíž, tím rychlejší může být dané zařízení)
  - Pokud si vzpomenete na velké boty jménem Meltdown a Spectre, tak tyto exploity fungovaly

načtením neautorizovaných instrukcí do mezipaměti
    - Mezipaměť často bývá SRAM a má několik úrovní. Prakticky každý procesor má L1 a L2 mezipaměť, většina také L3 a některé i L4
    - S každou úrovní zpravidla stoupá velikost paměti a klesá jejich rychlost, ale i tak L3 u standardních Intelů bude zhruba 2x rychlejší než RAM
    - **L1 -** mezipaměť nejbližší procesoru, bývá zpravidla nejrychlejší. Na rozdíl od ostatních úrovní byla zpravidla vždy zabudována do procesoru. Mívá až cca 64kb per jádro
    - **L2 -** druhotná mezipaměť, má až 512ḱb per jádro
    - **L3** - často poslední úroveň, bývá sdílená pro všechny jádra
    - **L4** - u procesorů, které ho mají, má velikost až 128mb, slouží spíš k prostému zrychlování (buffering) RAM
    - Někdy se můžete setkat také s mrňavou L0 mezipamětí. Konkrétní definice, umístění a účel se liší, zpravidla se však jedná o specializovanou mezipaměť. Někdy sedí mezi CU a L1 mezipamětí.
- **Jádro procesoru**
    - Spousta procesorů má více jader, to znamená, že může dělat více věcí najednou. Jinak totiž může procesor vykonávat jen jednu instrukci najednou a scheduler operačního systému přepíná mezi programy, aby se vytvořila iluze, že počítač dělá více věcí najednou
    - Pro každé jádro je zkopírována většina komponent
    - **Buyers beware:** U Intelu se můžete setkat s technologií zvanou Hyperthreading. Díky ní může jedno jádro mít dvě vlákna (tj. dělat dvě věci najednou). AMD má něco obdobného. Spousta lidí se však mylně domnívá, že mít dvě jádra s Hyperthreadingem (tj. mít dvě logická jádra) je srovnatelná jako mít čtyři fyzická jádra. Není to tak. Hyperthreading totiž funguje tak, že umožní využívat na druhém vláknu ty části procesoru, které to první nepoužívá. Obzvlášť pokud děláme něco náročnějšího, tak může být jádro využito celé a druhé vlákno čeká. V praxi je zisk výkonu jen zhruba 15-30% a to pouze při některých operacích. Usnadňuje to ovšem trochu práci operačnímu systému. Jenom taková zajímavost :)

Harvardská architektura

Harvardská architektura je model počítačové architektury (neplést s procesorovou architekturou). Její charakteristickou vlastností je to, že odděluje paměť na instrukce programu od paměti dat programu. To má jisté výhody i nevýhody. Na jednu stranu je efektivnější, může použít dva odlišné typy paměti, které se hodí pro daný účel (instrukce třeba budou v menší, rychlejší, ale dražší paměti, data v pomalejší, levnější a větší paměti). Na druhou stranu je ale složitější jí zrealizovat a není natolik flexibilní z hlediska programátora operačních systémů.

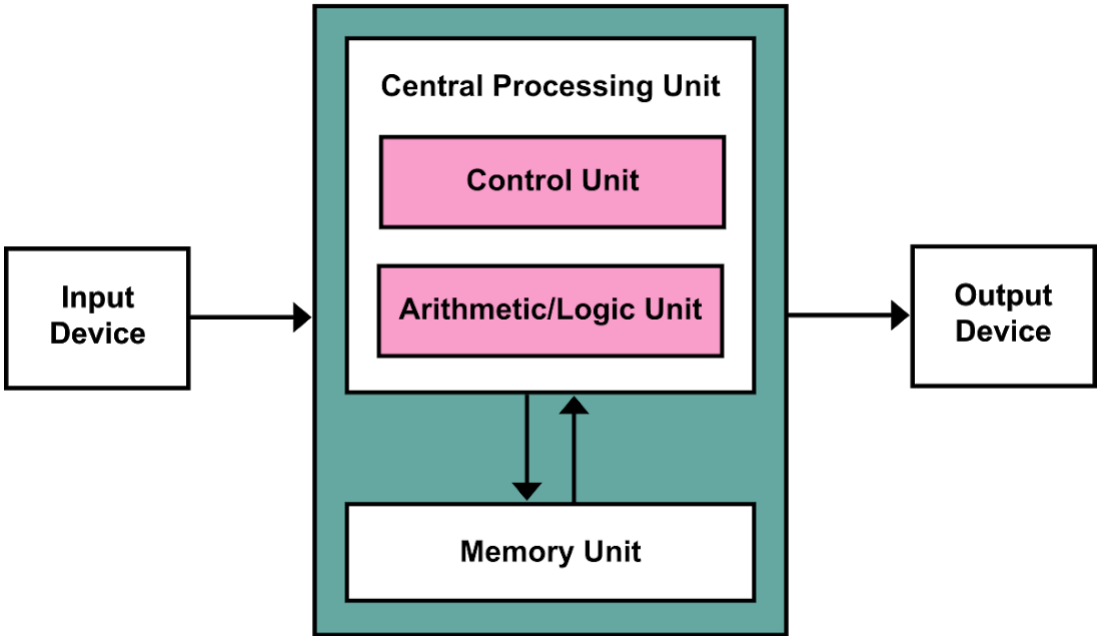
Tady je jednoduché schéma toho, co komunikuje s čím:



I/O znamená vstup/výstup.

Von Neumannova architektura
Tuto architekturu definoval John von Neumann. Narozdíl od harvardské používá jednu operační paměť jak pro instrukce, tak pro data. To umožňuje větší flexibilitu pro programátory, ale zase není tak efektivní a bývá trochu pomalejší z hlediska výkonu.

Schéma von Neumanna:



Toto schéma je sice vizuálně trochu složitější, ale jedná se o de facto to samé, až na rozdíl s pamětí.

Modifikovaná harvardská architektura
Když se moderní výrobci počítačů rozhodovali o tom, která architektura se dominantní, tak se na to někdo podíval zhruba takto:

A tak se i stalo. V současnosti dominantní modifikovaná harvardská architektura kombinuje obojí. Z vnější pohledu se duo CPU + RAM chová jako von Neumannova architektura, ale uvnitř procesoru (jádra + registry + mezipaměť) fungují na bázi harvardské architektury.

Procesorová architektura a RISC je ZISC
Procesory mají ovšem také vlastní architektury. Určují vnitřní fungování procesoru, instrukční sadu a další parametry. Ve stolních počítačích a laptopech se nejčastěji setkáte s architekturami x86 a x86_64 (někdy také x64 nebo amd64), první pro 32-bitové počítače adruhé pro 64-bitové. V telefonech bude ARM.

Architektury se dělí na dvě kategorie:

- **CISC** - **Complex Instruction Set Computing** - architektury, které mají velký počet instrukcí, například u x86_64 se najde až tři tisíce instrukcí, podle toho, jak to počítáte.
- **RISC** - **Reduced Instruction Set Computing** - tyto architektury mají poměrně málo instrukcí, třeba padesát, patří sem například RISC-V nebo MIPS

Assembly
Assembly velmi primitivní, nízkoúrovňový jazyk pro reprezentaci instrukcí pro procesor způsobem, aby to člověk přečetl (to neznamená, že tomu ale bude hned rozumět :^)). První programy pro každou architekturu a některé části jádra operační systému jsou zpravidla napsané v Assembly. Assembly není "portable", pro každou architekturu se liší. Proto se používají programovací jazyky vyšší úrovně, které se kompilují do patřičného strojového kódu, ale jsou jinak stejné pro všechny architektury.

Na hodině jsme si ukazovali tento demonstrační procesor: https://schweigi.github.io/assembler-simulator/index.html

Později ještě pošlu ukázkový program na absolutní rozdíl z minulé hodiny.

Zatím díky,
LH