B35APO: Architektury počítačů

Lekce 03. Central Processing Unit (CPU)

Pavel Píša
pisa@fel.cvut.cz

Petr Štěpán
stepan@fel.cvut.cz

14. března, 2023

Obsah

1 Procesor

2 Kódování instrukcí

3 Konstrukce procesoru

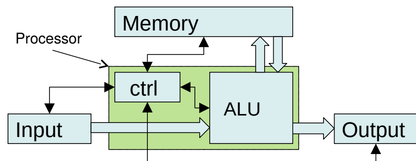
Simulátory QtMips a QtRVSim

- dříve se používalo prostředí MipsIt, které bylo ale velmi omezené
- QtMips se pro výuku APO využívá od letního semestru 2019
  - QtMIPS bylo vytvořeno jako diplomová práce Karla Kočího vedená Pavlem Píšou: *Graphical CPU Simulator with Cache Visualization*, kterou si můžete prohlédnout `https://dspace.cvut.cz/bitstream/handle/10467/76764/F3-DP-2018-Koci-Karel-diploma.pdf`
  - Opravy, rozšíření a částečné přepracování Pavel Píša
- v roce 2022 byl simulátor přepracován pro architekturu RISC-V, hlavní změny jsou od Jakuba Dupáka z jeho bakalářské práce z 2021 *Graphical RISC-V Architecture Simulator - Memory Model and Project Management* odkaz: `https://dspace.cvut.cz/bitstream/handle/10467/94446/F3-BP-2021-Dupak-Jakub-thesis.pdf`
- Alternativy:
  - RARS: Risc-V Assembler and Runtime Simulator – IDE založené na systému MARS `https://github.com/TheThirdOne/rars`
  - EduMIPS64: simulátor napsaný v Javě `https://www.edumips.org/`

QtRVSim - Download

- Windows, Linux, Mac
  https://github.com/cvut/qtrvsim/releases
- Ubuntu
  https://launchpad.net/~qtrvsimteam/+archive/ubuntu/ppa
- Suse, Fedora and Debian https://software.opensuse.org/download.html?project=home%3Ajdupak&package=qtrvsim
- Suse Factory TBD
- Online version https://dev.jakubdupak.com/qtrvsim/
- LinuxDays 2019 QtMips talk – záznam interaktivní přednášky
  https://youtu.be/fhcdYtpFsyw,
  https://pretalx.linuxdays.cz/2019/talk/EAYAGG/

John von Neumann



- 5 základních jednotek – řídicí jednotka, aritmeticko-logická jednotka, paměť, vstup (vstupní zařízení), výstup (výstupní zařízení)
- Architektura počítače by neměla být závislá na řešené úloze, měla by umět provádět program uložený v paměti. Program řídí, co počítač vykonává za instrukce a tím jaké dostane výsledky.
- Program a data jsou uložena ve stejné paměti, složené z buněk (jednotek) stejné velikosti. Oproti tomu Harvardská architektura měla jeden typ paměti pro program a jiný typ paměti pro data.
- Další instrukce, která se bude vykonávat je uložena v následující buňce paměti (mimo skoků v programu)
- Instrukce provádějí aritmetické a logické operace, přesuny dat z/do paměti, skoky a větvení programu a speciální řídicí instrukce.

Počítač založený na von Neumannově konceptu

Procesor / mikroprocesor:

- Řídicí jednota (control unit)
- Aritmeticko-logická jednotka – ALU

Paměť:

- von Neumann architektura má společnou paměť pro program i data
- paměť se skládá z buněk - jednotek, v současné době z bajtů

Vstupně výstupní podsystém:

- Vstupy – klávesnice, myši, děrné štítky, magnetopáskové jednotky, síťové karty, ...
- Výstupy – monitory (grafické karty), tiskárny, plotry, síťové karty, ...

Řídicí jednotka řídí práci a sekvenci této práce. Skládá se z:

- registrů – udržují mezivýpočty a stavy výpočtů
- řídicích hradel – dekódují instrukce a provádí operace

Nejdůležitější registry procesoru

- PC (Program Counter) – adresa právě prováděné (nebo následující) instrukce
- IR (Instruction Register) – obsahje kód prováděné instrukce načtený z paměti
- Another usually present registers
  - GPRs (General purpose registers) – obecné uživatelské registry, mohou se dělit na data a adresy do paměti, nebo být univerzální
  - SP (Stack Pointer) – ukazuje na vrchol zásobníku, slouží k organizaci lokálních dat funkcí
  - PSW (Program Status Word) – definuje v jakém stavu je procesor
  - IM (Interrupt Mask) – kontrola přerušení
  - FPRs (Floating point registers) – rozšíření procesoru pro práci s reálnými čísly, případně i vektorové/multimediální registry

Základní cyklus procesoru

1. Počáteční nastavení – inicializace registru PC a PSW (případně i dalších) po zapnutí proudu nebo po resetu procesoru
2. Přečti instrukci z paměti z adresy PC
   - nastav PC na adresu sběrnice paměti
   - Přečti obsah ze sběrnice paměti do registru IR
   - $PC + l \rightarrow PC$, uprav PC, l je délka načtené instrukce
3. Dekóduj instrukci
4. Proveď instrukci
   - spočti adresu, vyber registry, načti operandy, proveď požadovaný výpočet ALU a ulož výsledky
5. Zkontroluj zda není přerušení nebo vyjímka (podrobně v přednášce 9)
6. Opakuj od kroku 2

Compilation: C Assembler Machine Code

```c
/* ffs  as log2(x)*/
int x = 157;
int y = -1;

while(x != 0) {
  x = x / 2;
  y = y + 1;
}
```

```
_start:
    // int  x = 157;
  addi a0, zero, 157
    // int  y = -1;
  addi t1, zero, -1
    // while(x != 0) {
  beq a0, zero, done
loop:
    //   x = x / 2;
  srli  a0, a0, 1
    //   y = y + 1;
  addi t1, t1, 1
    // }
  bne a0, zero, loop
done:
```

```
0x00000200 09d00513
0x00000204 fff00313
0x00000208 00050863
0x0000020c 00155513
0x00000210 00130313
0x00000214 fe051ce3
0x00000218 00030513
0x0000021c 00100073
```

Obsah

1 Procesor

2 Kódování instrukcí

3 Konstrukce procesoru

Kódování instrukcí

Co musíme uvažovat při návrhu zakódování instrukcí?

- 8 bitů pro jednu instrukci
  - Pokud by velikost instrukce byla jeden bajt, tedy 8 bitů, tak můžeme mít maximálně 256 různých instrukcí
  - Je to dost?
  - Co vlastně musí být zakódováno uvnitř instrukce? Nutně tam musí být registry, se kterými bude instrukce pracovat
    - pokud bychom měli v procesoru 8 registrů, pak 3 bity kódují, který registr chceme použít
    - pro operaci potřebujeme 3 registry `cíl = zdroj1 + zdroj2` nebo jen 2 registry `cíl += zdroj`
    - pro práci s pamětí stačí 2 registry `cíl = MEM[zdroj]`
  - nelze zakódovat 3 operandy, i pro 2 operandy máme jen 2 bity (8-2*3) na zakódování typu instrukce, což je maximálně 4 druhy instrukcí
  - řešení by bylo jedině mít méně registrů a uvažovat zásobníkové operace (push, pop, sečti dvě čísla z vrcholu zásobníku, apod.)

Kódování instrukcí

- 16 bitů na jednu instrukci – 65536 různých instrukcí
  - lze 16 registrů, 256 instrukcí s dvě operandy, nebo 16 druhů instrukcí se třemi operandy ($4 + 3 * 4 = 16$ bitů)
- 32 bitů – přes 4 miliardy různých instrukcí
  - často 32 registrů, až 128 tisíc druhů instrukcí se třemi operandy ($17 + 3 * 5 = 32$ bitů)
- Další problém jsou konstanty, třeba chci přičíst 1 k registru, nebo uložit do registru konkrétní hodnotu 125.
  - překladač vygeneruje do paměti všechny konstanty a registry na ně budou ukazovat
  - nebo malá čísla budou součástí instrukce
    - toto je nejpoužívanější řešení
    - RISC má instrukce pevné délky a součástí 32-bitové instrukce může být 20-bitová nebo 12-bitová konstanta
    - CISC má instrukce proměnné délky a třeba i 64-bitové číslo je součástí kódu instrukce

RISC-V – Délka instrukce

base+4	base+2	base	
		`xxxxxxxxxxxxxxaa`	16-bit ($aa \neq 11$)
	`xxxxxxxxxxxxxxxx`	`xxxxxxxxxxxbbb11`	32-bit ($bbb \neq 111$)
`...xxxx`	`xxxxxxxxxxxxxxxx`	`xxxxxxxxxx011111`	48-bit
`...xxxx`	`xxxxxxxxxxxxxxxx`	`xxxxxxxxx0111111`	64-bit
`...xxxx`	`xxxxxxxxxxxxxxxx`	`xnnnxxxxx0111111`	$(80 + 16 \cdot nnn)$-bit ($nnn \neq 111$)
`...xxxx`	`xxxxxxxxxxxxxxxx`	`x111xxxxx0111111`	rezervováno pro $\geq$ 192-bit

Address:

base+4 base+2 base

RISC-V – Registry

Register	ABI Name	Popis	Uchování
x0	zero	má vždy hodnotu 0	
x1	ra	Návratová adresa (Return address)	Volající
x2	sp	Ukazatel na zásobník (Stack pointer)	Volaný
x3	gp	Ukazatel na globální data (Global pointer)	
x4	tp	Ukazatel na vlákno (Thread pointer)	
x5-7	t0–2	Dočasné proměnné (Temporaries)	
x8	s0/fp	Uchovávaný registr (Saved register)/ Ukazatel na rámec (Frame pointer)	Volaný
x9	s1	Uchovávaný registr (Saved register)	Volaný
x10–11	a0–2	Argumenty funkce (Function arguments)/ Návratové hodnoty (Return values)	Volající Volaný
x12–17	a2–7	Argumenty funkce (Function arguments)	Volající
x18–27	s2–11	Uchovávané registry (Saved registers)	Volaný
x28–31	t3–6	Dočasné proměnné (Temporaries)	
pc	pc	Ukazatel na instrukci (Program Counter)	
f0-31		Reálná čísla (Floating point)	

Cíl přednášky

Porozumět implementaci jednoduchého počítače, který se skládá z procesoru a oddělené paměti pro instrukce a data.

Naším cílem je implementovat následující instrukce:

- Číst a zapisovat hodnotu z/do datové paměti.
  - **lw** - načtení slova, **sw** - uložení slova
- aritmetické a logické instrukce: **add**, **sub**, **and**, **or**, **slt**
  - Immediátové varianty: **addi**, **ori**, horní bity načítají **olui**,**auipc**
- Instrukce pro změnu toku programu/skok **beq**
- Volání podprogramu **jal**, **jalr** (zajišťuje i návrat z podprogramu **jr ra**)
- CPU se bude skládat z řídicí jednotky a ALU.
- Poznámky:
  - Implementace bude minimální (jednocyklový procesor - všechny operace budou probíhat v jednom cyklu, tzn. zpracovány v jednom kroku/hodině)
  - Přednáška 5 je zaměřena na efektivnější realističtější implementaci zřetězenného (pipelined) CPU.

Kódování instrukcí procesoru MIPS

Starší typ procesoru, má tři typy instrukcí:

Typ	31 … 26	25 … 21	20 … 16	15 … 11	10 … 6	5 … 0
R	opcode(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)
I	opcode(6)	rs(5)	rt(5)	immediate(16)		
J	opcode(6)	address(26)				

- typ instrukce (R,I,J) je určen 6-bity opcode
- rs – register zdroje – source; rd – register cíle – destination; rt – někdy zdroj, někdy cíl
- immediate, address – konstanty použité pro operaci, immediate pro počítání, address pro skoky
- shamt – konstanta využitá pouze pro posuny (shift, operace <<, >>)
- funct – specifikace prováděné operace, např. sčítání, odčítání, posuny, atd.

Kódování instrukcí procesoru RISC-V

Má šest typů instrukcí:

<table>
<tr><th>Typ</th><th>31</th><th>30…25</th><th>24…21</th><th>20</th><th>19…15</th><th>14…12</th><th>11…8</th><th>7</th><th>6…0</th></tr>
<tr><td>R</td><td colspan="2">fnct7</td><td colspan="2">rs2</td><td>rs1</td><td>fnct3</td><td colspan="2">rd</td><td>opcode</td></tr>
<tr><td>I</td><td colspan="4">imm[11:0]</td><td>rs1</td><td>fnct3</td><td colspan="2">rd</td><td>opcode</td></tr>
<tr><td>S</td><td colspan="2">imm[11:5]</td><td colspan="2">rs2</td><td>rs1</td><td>fnct3</td><td colspan="2">imm[4:0]</td><td>opcode</td></tr>
<tr><td>B</td><td>imm [12]</td><td>imm [10:5]</td><td colspan="2">rs2</td><td>rs1</td><td>fnct3</td><td>imm[4:1]</td><td>imm [11]</td><td>opcode</td></tr>
<tr><td>U</td><td colspan="6">imm[31:12]</td><td colspan="2">rd</td><td>opcode</td></tr>
<tr><td>J</td><td>imm [20]</td><td colspan="2">imm[10:1]</td><td>imm [11]</td><td colspan="2">imm[19:12]</td><td colspan="2">rd</td><td>opcode</td></tr>
</table>

- délka a typ instrukce (R,I,S,B,U,J) je určen 7-bity opcode
- rs1,2 – registery zdroje – source; rd – register cíle – destination
  - registry jsou vždy na pevné pozici v instrukci
- immediate – konstanty použité pro operaci nebo skoky
- fnct3, fnct7 – specifikace prováděné operace, např. sčítání, odčítání, posuny, atd.

Kódování instrukcí procesoru RISC-V

Význam hodnot opcode:

Opcode	Skupina operací	Skutečná operace
0110011	Typ R podle fnct	add, sub, slt, or, and
0010011	Typ I podle fnct	addi, subi, slt, or, and
0000011	Čtení z paměti	lw
0100011	Zápis do paměti	sw
1100011	Větvení – branch	beq
1101111	Podprogramy	jal
1100111	Návrat z podprogramu	jalr
0000111	Načtení horní části konstanty	lui

Pro typ R jsou následující hodnoty `fnct3` a `fnct7`:

fnct7	fnct3	Skutečná operace
0000000	000	add
0100000	000	sub
0000000	010	slt
0000000	110	bitový or
0000000	111	bitový and

Bonusový bod

Jak lze v RISC-V načíst do registru konstantu o velikost 32-bitů?

A pomocí jedné instrukce

B pomocí dvou instrukcí, první načte horních 20 bitů, druhá spodních 12-bitů

C pomocí tří instrukcí, první načte 16 bitů, druhá provede bitový posun, třetí načte spodních 16 bitů

D pomocí pěti instrukcí, první načte 12 bitů, druhá provede bitový posun, třetí načte dalších 12 bitů, čtvrtá provede bitový posun a pátá načte nejnižších 8 bitů

Obsah

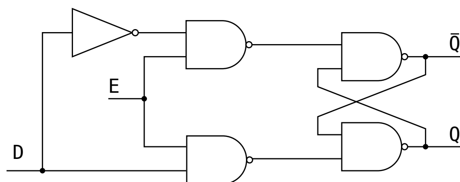
1 Procesor

2 Kódování instrukcí

3 Konstrukce procesoru

Hardwarová realizace registru

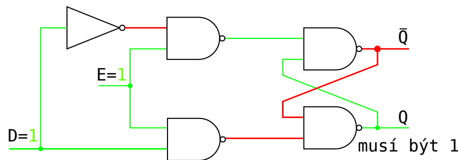
Základní realizace registru pomocí tzv. obvodu d-latch

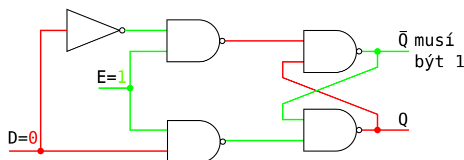


Tento obvod je velkou změnou oproti dosavadnímu přístupu, protože se zde objevuje cyklus – výstup hradla je vstupem stejného hradla, nebo vstupem hradla jehož výstup je vstupem prvního hradla.
Co to tedy dělá?

Hardwarová realizace registru

Chování obvodu je určeno hodnotou vstupu E.
Nejdříve se podíváme, co se děje při E=1:

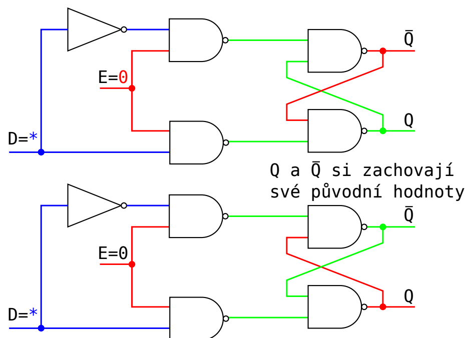




- Vstup D se se zpožděním objeví na výstupu Q
  - pro $D = 1$ musí být výstup $Q = 1$, protože jeden vstup do posledního nand hradla je 0, tedy výstup musí být 1
  - pro $D = 0$ musí být $\overline{Q} = 1$, protože opět jeden vstup do nand hradla je 0, když je $\overline{Q} = 1$, tak potom je výstup $Q = 0$, protože oba dva vstupy jsou 1, výstup nand je 0

Hardwarová realizace registru

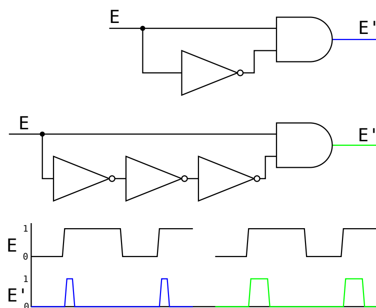
Pokud je E=0:



- Hodnota $Q, \overline{Q}$ závisí pouze na staré hodnotě dvojice $Q, \overline{Q}$
- Dvojice nabývá pouze hodnot, která mohla vzniknout při zápisu, tedy buď $Q = 1, \overline{Q} = 0$ nebo $Q = 0, \overline{Q} = 1$

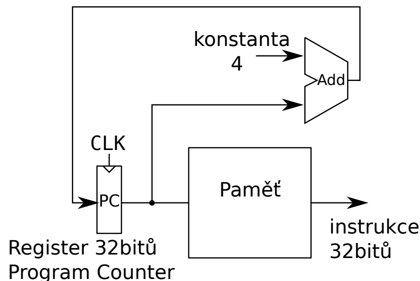
Hardwarová realizace registru

Aktivace náběžnou hranou:



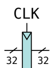
- Může být výstup tohoto obvodu vůbec někdy v 1?
- matematicky je vždy $(C \text{ and } \overline{C}) = 0$
- ve skutečnosti je ale hradlo not zpožděno oproti normálnímu signálu a na tuto dobu zpoždění je výstup `E'` v 1
- pokud by tato hodnota byla moc krátká na to, aby se obvod d-latch nastavil, tak je možné přidat negací více za sebou (zelená křivka)
- tomuto obvodu se říká d flip-flop

Hardwarová realizace základního cyklu CPU

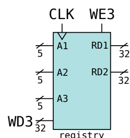


Při náběžné hraně hodin (signál CLK) se do registru PC uloží nová hodnota PC+4. Tím se automaticky rozběhne nové čtení z paměti a následné zpracování instrukce, které si ukážeme později.

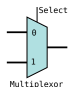
Stavební bloky procesoru



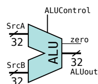
jeden registr, zápis řízen hodinovým signálem CLK



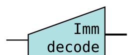
soubor 32 registrů, podle vstupu A1, A2 vybere hodnoty těchto dvou registrů na výstup RD1 a RD2; pokud přijde hodinový signál a je nastaven vstup WE3 (write enable) tak se hodnota WD3 zapíše do registru číslo A3



na základě hodnoty řídicího signálu Select přenáší hodnotu jednoho ze vstupů na výstup

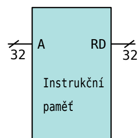


na základě ALUControl provede vybranou operaci na vstupu SrcA a SrcB. Výsledek je ALUout a výsledkem jsou i příznaky jak vyšel výsledek, například zero

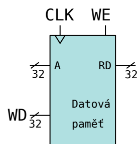


provede sestavení bitů konstanty z bitů instrukce a provede znaménkové rozšíření na 32 bitů plnohonotného čísla se znaménkem

Stavební bloky procesoru



Instrukční paměť dá na výstup data RD uložená v paměti na adrese A. Data jsou k dispozici po uplynutí "dostatečně dlouhé"doby pro nalezení dat a jejich vystavení na výstup RD.



Datová paměť. Slouží buď ke čtení dat z adresy A. Protože se jedná o stejnou pamět jako je instrukčí, tak po "dostatečně dlouhé"době jsou data na výstupu RD. Pokud je nastaven příznak WE (write enable), tak s náběžnou hranou hodinového signálu CLK dojde k zápisu dat WD na adresu A.

Instrukce lw – load word

`lw -- load word` načti slovo z paměti do registru

Popis	Načte slovo do registru ze zadané adresy z paměti
Operace	[rd] ← Mem[[rs1]+imm12]
Syntax	lw rd, imm12(rs1)
Kódování	iiii iiii iiii ssss s010 dddd d000 0011
	s – rs1; d – rd; i – konstanta

Příklad: Načtěte 32-bitové slovo z paměti z adresy 0x400 do registru 2:

`lw x2, 0x400(x0)`

```
iiii iiii iiii  ssss s010  dddd d000 0011
0100 0000 0000  0000 0010  0001 0000 0011
```

0x400 | 0 | *func3* | 2 | *opcode*

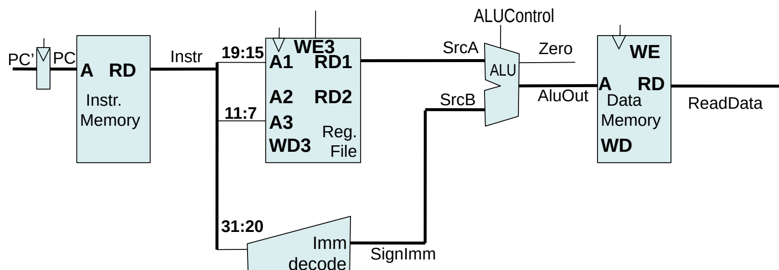
`0x 40 00 21 03` – kód pro instrukci `lw x2, 0x400(x0)`

Poznámka: Registr x0 má natvrdo hodnotu 0, kterou nelze změnit

Implementace instrukce lw

`lw`: `rs1` – základní adresa, `imm12` – posunutí adresy, `rd` – registr, kam se uloží data z paměti

Typ	31	30…25	24…21	20	19…15	14…12	11…8	7	6…0
I	imm[11:0]				rs1	fnct3	rd		opcode

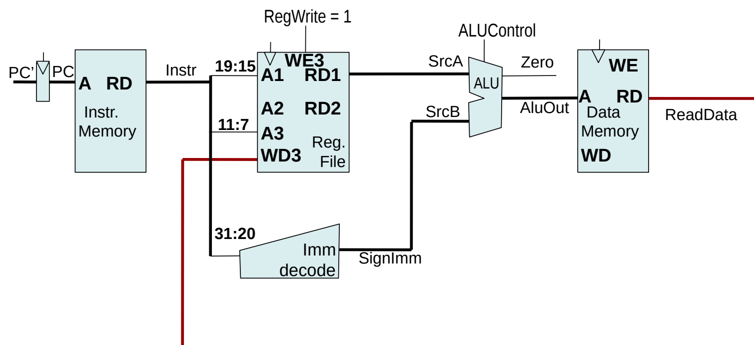


Implementace instrukce lw

`lw`: `rs1` – základní adresa, `imm12` – posunutí adresy, `rd` – registr, kam se uloží data z paměti

Typ	31	30…25	24…21	20	19…15	14…12	11…8	7	6…0
I	imm[11:0]				rs1	fnct3	rd		opcode

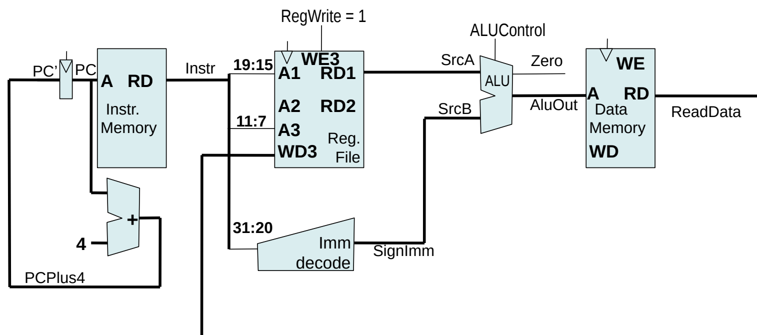
Zápis proběhna na rostoucí hranu hodin.



Implementace instrukce lw

`lw`: `rs1` – základní adresa, `imm12` – posunutí adresy, `rd` – registr, kam se uloží data z paměti

Typ	31	30…25	24…21	20	19…15	14…12	11…8	7	6…0
I	imm[11:0]				rs1	fnct3	rd		opcode



QtRvSim - RISC-V Simultaor

Load

Run

Single Step

Make

Exceptions control

Registers

Terminal

Assembler

Editor

Peripherals

Code

CPU core view
- single cycle
- pipelined

Cache

Data memory

Instrukce sw – store word

`sw -- store word` ulož slovo z registru do paměti

Popis	Uloží slovo z registru rs2 do zadané adresy z paměti
Operace	`Mem[[rs1]+imm12]` ← `[rs2]`
Syntax	sw rs2, imm12(rs1)
Kódování	`iiii iiit tttt ssss s010 iiii i010 0011` t – rs2; s – rs1; i – konstanta

Příklad: Uloží 32-bitové slovo z regisrtu 2 do paměti na adresu 0x404 plus hodnota registru 5:

`sw x2, 0x404(x5)`

```
iiii iiit tttt ssss s010 iiii i010 0011
0100 0000 0010 0010 1010 0010 0010 0011
```

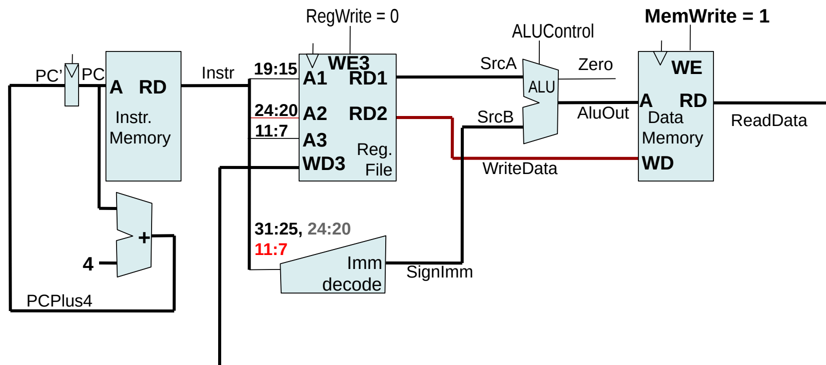
0x40_ | 2 | 5 | *func3* | 0x_04 | *opcode*

`0x 40 22 a2 23` – kód pro instrukci **`sw x2, 0x404(x5)`**

Implementace instrukce sw

sw: rs1 – základní adresa, imm12 – posunutí adresy, rs2 – registr, který se uloží do paměti

Typ	31	30…25	24…21	20	19…15	14…12	11…8	7	6…0
S	imm[11:5]		rs2		rs1	fnct3	imm[4:0]		opcode



Instrukce add – sečti dva registry

`add -- addition` – sečti hodnoty dvou registrů a součet ulož do registru

Popis	Sečti hodnoty z registrů rs1 a rs2 a výsledek ulož do rd
Operace	`[rd]` ← `[rs1] + [rs2]`
Syntax	add rd, rs1, rs2
Kódování	`0000 000t tttt ssss s010 dddd d011 0011` t – rs2; s – rs1; d – rd

Příklad: Sečti hodnoty z registru 2 a registru 3, výsledek ulož do registru 4:

`add x4, x2, x3`

```
0000 000t tttt ssss s000 dddd d011 0011
0000 0000 0011 0001 0000 0010 0011 0011
```

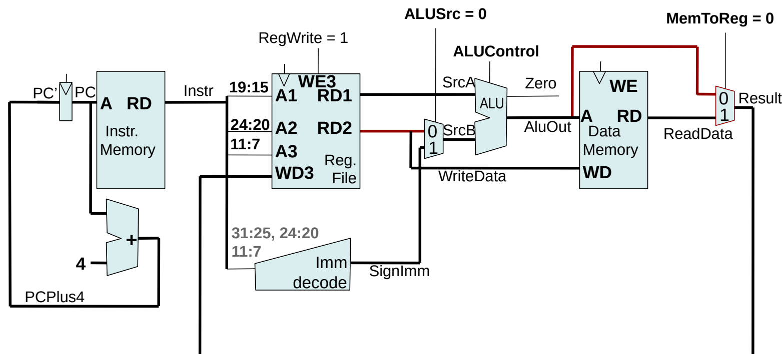
0000 000	0 0011	0001 0	000	0010 0	011 0011
funct7	3	2	*func3*	4	*opcode*

`0x 00 31 02 33` – kód pro instrukci **`add x4, x2, x3`**

Implementace instrukce add

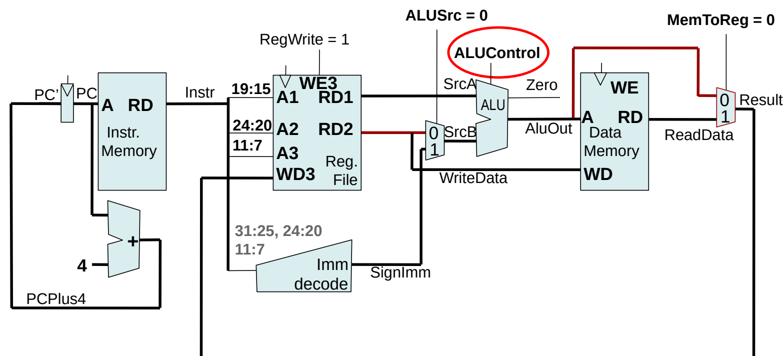
`add`: `rs1`, `rs2` – sčítance, `rd` – registr, kam se uloží součet

Typ	31	30...25	24...21	20	19...15	14...12	11...8	7	6...0
R	fnct7		rs2		rs1	fnct3	rd		opcode



Implementace instrukce sub, amd, or, slt

Jediný rozdíl oproti instrukci sčítání add je v hodnotě ALUControl, který vybírá, jakou operaci jednotka ALU provede. Cesta dat a uložení výsledku je stejná jako u sčítání.



Instrukce addi – přičti konstantu k registru

addi -- addition immediate – sečti hodnotu registru a konstanty a výsledek ulož do registru

Popis	Sečti hodnoty z registru rs1 a imm12 a výsledek ulož do rd
Operace	`[rd] ← [rs1] + imm12`
Syntax	addi rd, rs1, imm12
Kódování	`iiii iiii iiii ssss s000 dddd d001 0011` i – konstanta; s – rs1; d – rd

Příklad: Zvětši hodnotu registru 7 a 4 (výsledek tedy ulož do registru 7):

addi x7, x7, 4

```
iiii iiii iiii  ssss s 000  dddd d 001 0011
0000 0000 0100  0011 1 000  0011 1 001 0011
```

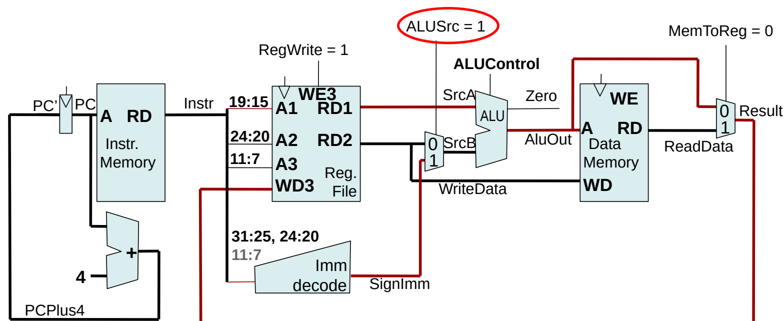
0x004 | 7 | *func3* | 7 | *opcode*

0x 00 43 83 93 – kód pro instrukci **addi x7, x7, 4**

Implementace instrukcí addi, ori, andi

`addi -- add immediate`: `[rd]` ← `[rs1] + imm12` – přičte k hodnotě registru `rs1` konkrétní číslo `imm` a výsledek uloží do registru `rd`

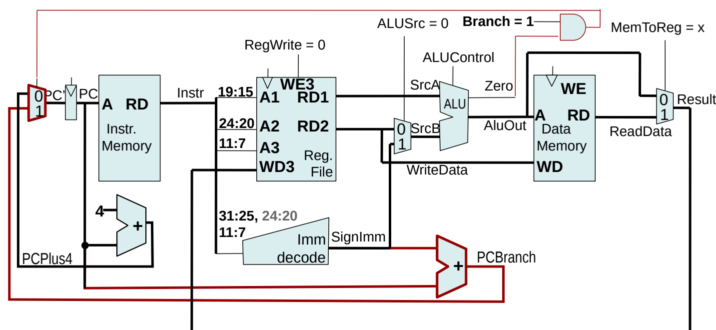
Typ	31	30...25	24...21	20	19...15	14...12	11...8	7	6...0
I	imm[11:0]				rs1	fnct3	rd		opcode



Implementace instrukce skoku beq

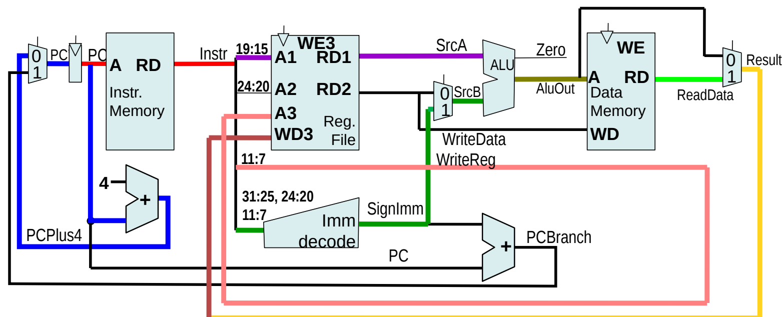
`beq -- branch if equal`: `[pc]` ← `[pc] + SignImm` – přičte k hodnotě registru pc konkrétní znaménkové číslo rozšířené na 32 bitů `imm` a výsledek uloží do registru pc pokud nastala rovnost hodnot v registrech rs1 a rs2.

Typ	31	30…25	24…21	20	19…15	14…12	11…8	7	6…0
B	imm [12]	imm [10:5]	rs2		rs1	fnct3	imm[4:1]	imm [11]	opcode



Rychlost CPU

$$T_c = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup}$$



Rychlost CPU

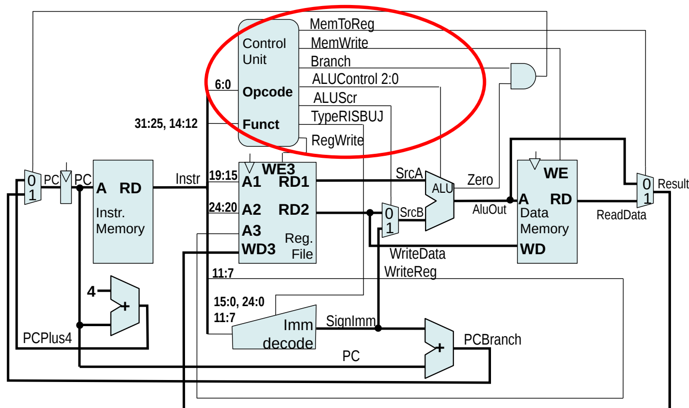
Jaká je maximální možná rychlost navrženého procesoru?

- Musí proběhnout všechny barevně označené fáze procesoru pro všechny instrukce a nejhorší případ je instrukce `lw`:
  - $t_{PC} = 0,3$ ns
  - $t_{Mem} = 20$ ns
  - $t_{RFread} = 1,5$ ns
  - $t_{ALU} = 2$ ns
  - $t_{Mux} = 0,1$ ns
  - $t_{RFsetup} = 0,1$ ns
- Celkem se dostáváme na čas $T_{CLK} = t_{PC} + t_{Mem} + t_{RFread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{RFsetup} = 44$ ns
- Tomuto času odpovídá frekvence $f_{CLK} = \frac{1}{T_{CLK}} = 22,7$ MHz tedy 22 700 000 instrukcí za sekundu = 22,7 MIPS
- Potřebovali bychom tedy procesor zrychlit, téma pro přednášku 5

Řídicí jednotka – Řadič procesoru

Co vlastně dělá řídicí jednotka (někdy také nazývaná řadič procesoru, angl. Control Unit)?

Převádí bity **`opcode`** a **`fnct3,7`** na řídicí bity ALUControl, RegWrite, MemWrite, ALUSrc, MemToReg, Branch a pár dalších pro další instrukce.



Řídicí jednotka – Řadič procesoru

Podle Opcode lze nadefinovat výstupy řadiče:

Instrukce	Opcode	Fnct3	Fnct7	ALUControl	ALUSrc	RegWrite	MemWrite	MemToReg	Branch
lw	0000011	010	–	+	1	1	0	1	0
sw	0100011	010	–	+	1	1	1	0	0
add	0110011	000	0000000	+	0	1	0	0	0
sub	0110011	000	0100000	-	0	1	0	0	0
slt	0110011	010	0000000	<<	0	1	0	0	0
or	0110011	110	0000000	|	0	1	0	0	0
and	0110011	111	0000000	&	0	1	0	0	0
addi	0010011	000	–	+	0	1	0	0	0
beq	1100011	000	–	==	0	0	0	*	1

Možné realizace řadiče

- Řadič obvodový:
  - Kombinační obvod (to byl náš případ),
  - Sekvenční obvod – stavový automat, apod.
- Řadič mikroprogramovaný (řízený mikroprogramem):
  - Mikroprogram je uložen v řídicí paměti řadiče a skládá se z mikroinstrukcí.
  - Mikroprogram implementuje programátorovi viditelné strojové instrukce (add, sub, lw, xor, jmp,. . .).
  - Operační kód instrukce udává adresu první mikroinstrukce v řídicí paměti, od které začíná mikroprogram pro danou instrukci.
  - Každá z instrukcí ISA je provedena pomocí jedné nebo několika mikroinstrukcí.
  - Výhodou mikroprogramovaného řadiče je flexibilita: změna mikroprogramu = změna chování procesoru.
  - Nevýhody: složitější a nevhodný pro zřetězené procesory, kdy každý stupeň vykonává jinou instrukci. Bylo by potřeba zajistit správné provedení všech instrukcí napříč stupni spolu s řešením hazardů

Bonusový bod

Které bity instrukce rozhodují o tom, jestli se bude ukládat do paměti, nebo počítat s dvěmi registry a výsledek se uloží do třetího registru?

A opcode

B fnct3,7

C rs1, rs2

D rd