

Architektura počítačových systémů (BI-APS), Přednáška č.3

Návrh jednocyklové RISC mikroarchitektury

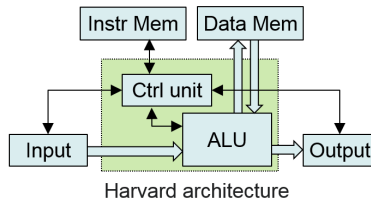
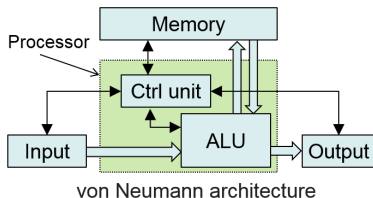
Michal Štepanovský, Pavel Tvrdlík



Czech Technical University in Prague
Faculty of Information Technology
<https://courses.fit.cvut.cz/BI-APS>

2022

(Version: 3. 10. 2022)



- 5 funkčních jednotek: řídicí jednotka (řadič), aritmeticko-logická jednotka, paměť, vstupní zařízení, výstupní zařízení.
- Nezávislost struktury počítače na zpracovávaných problémech. Musí se zavést program a musí se uložit do paměti. Ten řídí činnost počítače.
- Paměť programu a dat:
 - ▶ **unifikována** \Rightarrow von Neumann
 - ▶ **separátní** \Rightarrow Harvard
- Paměť je rozdělena na stejně velké části (buňky), které jsou průběžně očíslované (adresa).
- Po sobě jdoucí instrukce se ukládají za sebou.
- Existují instrukce aritmeticko-logické, přenosu dat (mezi CPU a pamětí, mezi CPU a V/V), řídicí a ostatní.

Úkol pro tuto přednášku I

- Dnešní počítače mají unifikovanou hlavní paměť a výhody CPU typicky používají na nejnižší úrovni paměťové hierarchie L1 instrukční cache a L1 datovou cache (architektura Harvard).
- Proto úkolem pro tuto přednášku bude

navrhnout **mikroarchitekturu** jednoduchého počítače tvořeného **procesorem** a **oddělenými pamětmi instrukcí a dat**.

- Na této přednášce budeme mikroarchitekturu navrhovat jednoduše jako **jednocyklovou**¹.
- V příští přednášce se budeme věnovat realističtější a zároveň o něco složitější **zřetězené mikroarchitektuře**.

¹Jednocyklová mikroarchitektura vykoná každou instrukci během trvání jednoho hodinového taktu.

Proč RISC-V?

- Zvyšující se poptávka po procesorech na míru (z pohledu výkonnosti a spotřeby).
- Hojně používané ISA (x86, x86-64, ARM, a další) jsou **intelektuálním vlastnictvím** daných společností. Například, pokud bychom chtěli navrhnout ARM-kompatibilní procesor, musí být licencován společností Arm Ltd., a za licenci bude nutné platit.
- RISC-V je otevřená a volně dostupná ISA (žádné poplatky za používání).
- RISC-V se drží principů RISC.
- RISC-V je **rodina spolu souvisejících ISA**.
- RISC-V je proto modulární/parametrizovatelná a uživatelsky rozšiřitelná. Hodí se nejenom pro malé počítače (vestavné systémy apod.), ale také pro vysoce výkonné počítačové systémy.
- Dokonce jsou dány k dispozici volně dostupné mikroarchitektury procesorových jader.
- RISC-V je jednoduchá a tedy vhodná pro výuku.



- **XLEN**: šířka celočíselných registrů v bitech (32 nebo 64)
- **I/E**: integer/embedded (RV32E používá pouze 16 registrů)
- **EXTENSIONS**:
 - ▶ **M**: Multiplication and division instructions
 - ▶ **A**: Atomic instructions
 - ▶ **F**: Single-Precision Floating-Point support
 - ▶ **D**: Double-Precision Floating-Point support
 - ▶ **C**: Compressed instructions (tj. kratší instrukce)
 - ▶ **Zicsr**: Control and Status Register instructions
 - ▶ **Zifencei**: Instruction-Fetch Fence
 - ▶ etc.
- Například, **RV64IMAFDZicsrZifencei** je 64-bitová ISA podporující základní celočíselné, rozšířené celočíselné a floating-point výpočty, a synchronizační primitiva pro více-jádrové výpočty, zatímco **RV32EC** je 32-bitová ISA vhodná pro vestavné systémy podporující pouze základní celočíselné instrukce s možností kódování instrukcí na 16 bitů (pokud možno).

Úkol pro tuto přednášku II

Jako základ si zvolíme **RV32I**:

- Základní 32-bitová celočíselná ISA s univerzálními registry (GPR).
- Registrů je 32, jsou celočíselné a jsou 32-bitové. Názvy registrů jsou: x0-x31.
- V registru x0 je trvale uložena konstanta 0.
- ALU instrukce jsou 3-adresní: ISA (3,0).
- Jako cílový nebo zdrojový registr může být použit libovolný z registrů x0-x31
=> princip GPR.
- Čísla zdrojových registrů (rs1, rs2) a číslo cílového registru (rd) jsou vždy zakódována na stejných bitech instrukce => zjednodušuje dekódování.
- Load/Store architektura (operandy ALU nemůžou přímo do/z paměti)
=> zjednodušuje návrh procesoru.
- Existují pouze 4 typy instrukcí (R/I/S/U) (+ 2 další typy (B/J) lišící se pouze v přímém operandu) => zjednodušuje dekódování.
- Všechny instrukce mají stejnou délku (32 bitů) => zjednodušuje dekódování.

Úkol pro tuto přednášku III

Instrukce našeho procesoru (podmnožina RV32I):

- Čtení a zápis do datové paměti Load a Store: `lw` a `sw`.
- Aritmeticko-logické instrukce `add`, `addi`, `sub`, `and`, `or` a `sll`.
- Instrukce podmíněného skoku `beq`.
- Instrukce volání podprogramu `jal` a `jalr`².

Tuto podmnožinu instrukcí budeme v rámci BI-APS označovat jako **picoRISC-V**. Pomocí těchto několika málo instrukcí lze psát zajímavé programy.

²Tyto instrukce v sobě rovněž zahrnují funkcionalitu instrukcí bezpodmíněného skoku `j` (jump) a `jr` (jump register) a návratu z podprogramu `ret` (return). Proto tyto instrukce (`j`, `jr` a `ret`) není potřeba implementovat.

Podporované instrukce

Syntaxe	Sémantika
lw rd, imm _{11:0} (rs1)	$rd \leftarrow \text{Mem}[[rs1] + \text{imm}_{11:0}];$
sw rs2, imm _{11:0} (rs1)	$\text{Mem}[[rs1] + \text{imm}_{11:0}] \leftarrow [rs2];$
addi rd, rs1, imm _{11:0}	$rd \leftarrow [rs1] + \text{imm}_{11:0};$
add rd, rs1, rs2	$rd \leftarrow [rs1] + [rs2];$
sub rd, rs1, rs2	$rd \leftarrow [rs1] - [rs2];$
and rd, rs1, rs2	$rd \leftarrow [rs1] \& [rs2];$
or rd, rs1, rs2	$rd \leftarrow [rs1] [rs2];$
slt rd, rs1, rs2	$rd \leftarrow [rs1] < [rs2];$
beq rs1, rs2, imm _{12:1}	if $[rs1] == [rs2]$ go to $[PC] + \{\text{imm}_{12:1}, '0'\}$; else go to $[PC] + 4$;
jal rd, imm _{20:1}	$rd \leftarrow [PC] + 4$; go to $[PC] + \{\text{imm}_{20:1}, '0'\}$;
jalr rd, rs1, imm _{11:0}	$rd \leftarrow [PC] + 4$; go to $[rs1] + \text{imm}_{11:0}$;

rd = register destination, **rs1(2)** = register source **1(2)**, **imm** = immediate operand.

Zápis *imm_{high:low}* u přímého operandu *imm* značí pozici bitů, které generuje ve výsledné 32-bitové přímé hodnotě. Přímý operand *imm* je doplněn nulami zprava, pokud dolní bity nejsou specifikovány, a poté znaménkově rozšířen na 32 bitů. Doplnění nulami je explicitně indikováno pomocí *{imm_{high:low}, padding}* v sémantice instrukce.

Názvy registrů RISC-V

K registrům lze přistupovat pomocí jejich architekturálního pojmenování (x0...x31) nebo pomocí jejich ABI³ jména.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

³Application binary interface (ABI) je nízkoúrovňové rozhraní mezi dvěma programy.

Ukázkové programy I

Součet hodnot v registrech:

```
addi x1,x0,4      //  $x1 \leftarrow 4$ ;  
addi x2,x0,20     //  $x2 \leftarrow 20$ ;  
add  x3,x1,x2     //  $x3 \leftarrow [x1] + [x2]$ ;
```

Inkrementace obsahu paměti na adrese 12:

```
lw    x1,12(x0)   //  $x1 \leftarrow \text{Mem}[12]$ ;  
addi  x1,x1,1     //  $x1 \leftarrow [x1] + 1$ ;  
sw    x1,12(x0)   //  $\text{Mem}[12] \leftarrow [x1]$ ;
```

Podmíněné přiřazení při nerovnosti obsahu registrů:

```
beq   x1,x2,L1    // if  $[x1] == [x2]$  go to L1;  
addi  x2,x0,5     //  $x2 \leftarrow 5$ ; (Assigned only if  $[x1] \neq [x2]$ )  
L1:
```

Ukázkové programy II

Volání podprogramu/rutiny (gcd = největší společný dělitel):

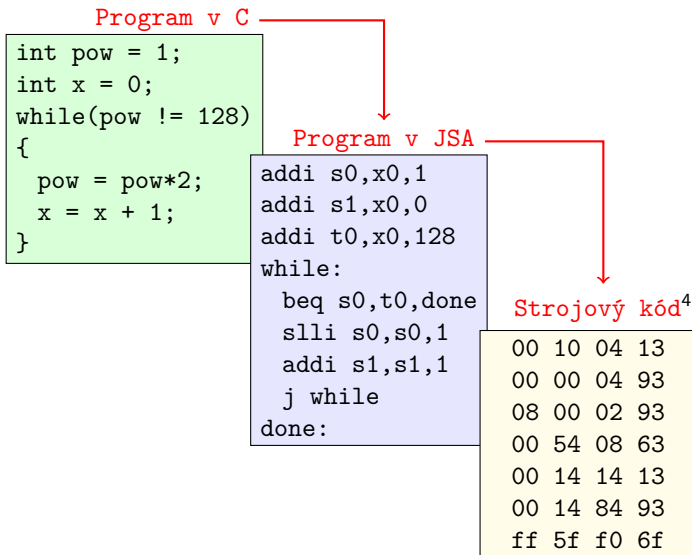
```
int gcd (int n1, int n2){
    while(n1!=n2){
        if(n1 > n2)
            n1 -= n2;
        else
            n2 -= n1;
    }
    return n1;
}

void main(){
    register int n1 = 25;
    register int n2 = 15;
    gcd(n1, n2);
}
```

```
gcd:
    beq a0,a1,done // Are we done?
    slt t0,a0,a1   // t0 ← [a0]<[a1];
    beq t0,x0,L    // [a0]<[a1]?
    sub a1,a1,a0   // a1 ← [a1]−[a0];
    beq x0,x0,gcd  // go to gcd;
L:sub a0,a0,a1    // a0 ← [a0]−[a1];
    beq x0,x0,gcd  // go to gcd;
done:
    jalr x0,x1,0   // return;

main:
    addi a0,x0,25 // a0 ← 25; (n1)
    addi a1,x0,15 // a1 ← 15; (n2)
    jal  x1,gcd   // Call gcd subroutine
```

Kompilace a kódování programu



⁴Strojový kód je vypsan hexadecimálně.

Formát instrukcí – základní RISC-V ISA (RV32I)

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20 10:1 11 19:12]										rd		opcode		J-type

- **opcode** = operační kód, **funct** = doplňující informace.
- **rs1**, **rs2** = source register, **rd** = destination register, **imm** = přímý operand.
- Typy B a J se liší od předchozích S a U pouze ve způsobu, jak je sestaven přímý operand (z důvodu optimalizace HW při dekódování).

Formát instrukcí – základní RISC-V ISA (RV32I)

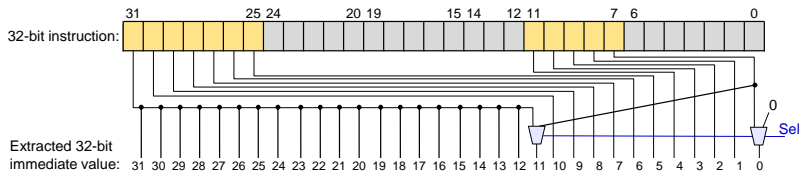
Porovnání formátů S a B:

31	25	24	20	19	15	14	12	11	7	6	0	
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12:10:5]		rs2		rs1		funct3		imm[4:1:11]		opcode		B-type

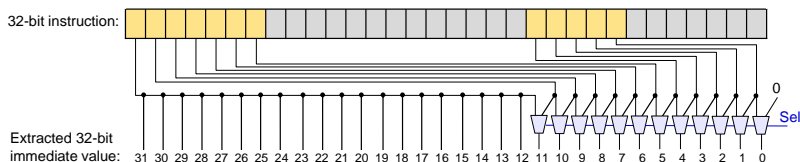
- Jediný rozdíl mezi formáty S a B je v 12-bitovém přímém operandu, kdy ve formátu B je vyjádřen v násobcích 2 (používá se pro zakódování PC-relativního offsetu skokové instrukce).
- Proto ve formátu B není imm[0] uváděn (je vždy 0), a naopak, imm[12] musí být specifikován.
- Aby se udržela cena HW minimální, co nejvíc bitů přímého operandu musí zůstat na stejné pozici.
- První možností by bylo zakódovat imm[12] namísto imm[0], tj. na bitu 7 instrukce (inst[7]).
- Nicméně, znaménkový bit musí být zakódován na pozici inst[31].
- Proto imm[12] je umístěn na pozici inst[31] a tím pádem imm[11] na pozici původního imm[0], tedy inst[7].

Formát instrukcí – základní RISC-V ISA (RV32I)

Získání S-immediate a B-immediate hodnot z instrukce:



Tradiční přístup by používal pouze jeden instrukční formát a posun v HW:



- Bitový posun v HW (násobení 2) vyžaduje víc multiplexorů.
- Toto je mnohem patrnější, pokud uvažujeme i další formáty (například U-type, který má 20-bitový přímý operand).
- Zvolené kódování přímého operandu v RISC-V nijak zásadně neovlivňuje kompilaci programů.

Formát instrukcí – Přímý operand

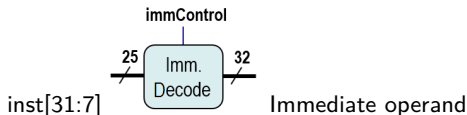
Jak z výše uvedené tabulky vyplývá, existuje 5 způsobů, jak z daných bitů instrukce vytvořit přímý operand:

31							0	
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]	I-immediate	
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]	S-immediate	
— inst[31] —				inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —				U-immediate	
— inst[31] —			inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

Poznámka: Pozice jednotlivých bitů ve variantách U a J jsou voleny tak, aby se co nejvíc překrývaly s již stávajícími typy a zároveň, aby znaménkový bit operandu byl vždy zakódován na nejvyšším bitu instrukce.

Formát instrukcí – Přímý operand

- Formát instrukce (R-type, I-type, atd.) je jednoznačně určen operačním kódem ($\text{inst}[6:0]$).
- Zbývající bity instrukce $\text{inst}[31:7]$ jsou použity pro získání přímého operandu (ne nutně všechny pro danou instrukci).
- V našem návrhu budeme používat následující jednotku pro získání přímého operandu:



Která je to instrukce?

Instrukce je určena především pomocí **opcode**:

opcode	Význam	Pro náš specifický případ
0110011	R-type (see funct7 and funct3)	add, sub, slt, or, and
0010011	ALU-imm (see funct3)	addi
0000011	Memory load (see funct3)	lw
0100011	Memory store (see funct3)	sw
1100011	Branch (see funct3)	beq
1101111	jal	jal
1100111	jalr	jalr

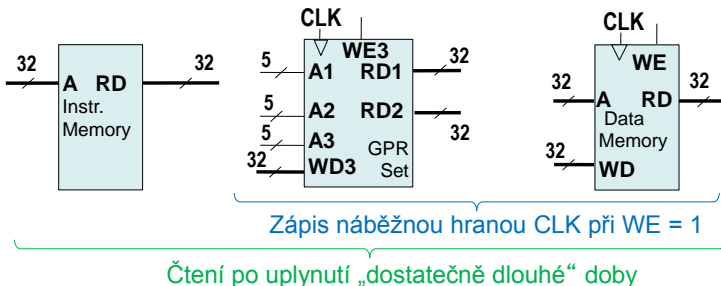
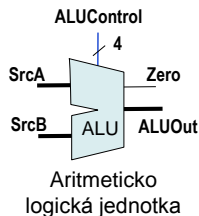
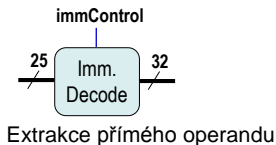
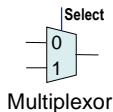
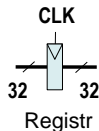
Pro instrukce typu R platí dále tabulka:

funct7	funct3	Instrukce
0000000	000	add
0100000	000	sub
0000000	010	slt
0000000	110	or
0000000	111	and

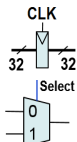
Formát a kódování jednotlivých instrukcí

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
imm[11:0]						rs1		010		rd		0000011	lw	
imm[11:5]			rs2			rs1		010	imm[4:0]			0100011	sw	
0000000			rs2			rs1		000	rd			0110011	add	
0100000			rs2			rs1		000	rd			0110011	sub	
0000000			rs2			rs1		010	rd			0110011	slt	
0000000			rs2			rs1		110	rd			0110011	or	
0000000			rs2			rs1		111	rd			0110011	and	
imm[11:0]						rs1		000	rd			0010011	addi	
imm[12 10:5]			rs2			rs1		000	imm[4:1 11]			1100011	beq	
imm[20 10:1 11 19:12]										rd			1101111	jal
imm[11:0]						rs1		000	rd			1100111	jalr	

Stavební prvky pro jednocyklový procesor

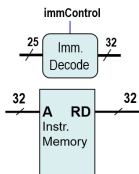


Stavební prvky pro jednocyklový procesor - popis

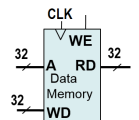


Registr ovládaný náběžnou hranou hodin CLK. Při náběžné hraně hodin přenáší hodnotu vstupu na výstup. Hodnota výstupu se pamatuje až do následující náběžné hrany hodin.

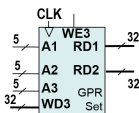
Multiplexor na základě řídicího signálu Select přenáší hodnotu jednoho ze vstupů na výstup.



Sestavení přímého operandu z instrukce v závislosti od jejího typu. Vstupem je instrukce, výstupem přímý operand. Viz slajd 16.









Instrukční paměť na portu RD vydá instrukci odpovídající vstupní adrese A. Čtení z paměti je kombinační.



Dvou-portová datová paměť. Port RD slouží pro čtení (kombinační), port WD slouží pro zápis dat. Zápis se provede pokud je povolen (řídicí vstup WE) a to pouze na náběžné hraně hodin (CLK).

Tří-portový soubor 32 registrů. Všechny registry jsou 32-bitové. Čtecí porty RD1 a RD2 jsou adresovány vstupy A1 a A2. Čtení je kombinační. Port pro zápis WD3 se adresuje vstupem A3. Zápis do vybraného registru se provede pokud je povolen (řídicí vstup WE3) a to na pouze náběžné hraně hodin (CLK).

Značení cest

	Tlustá (černá) čára: 32-bitová datová cesta (= 32 paralelně vedených vodičů).
	Tenká (černá) čára: Datová cesta jiné šíře než 32 bitů.
	Tenká modrá čára: Řídicí signál přicházející z řídicí jednotky.
	Křížení vodičů (vodiče nejsou galvanicky spojeny).
	Rozdvojení cesty (vodiče jsou galvanicky spojeny).
	Výběr bitů 15:0 z datové cesty.

Výklad syntaxe a sémantiky instrukce: například `lw`

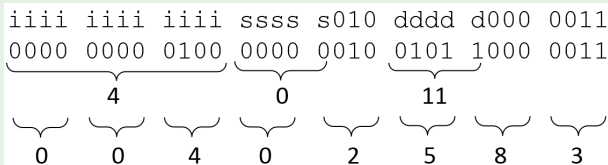
`lw` = load word - čtení slova z datové paměti

Description	A word is loaded into a register from the specified address
Operation:	$rd \leftarrow \text{Mem}[[rs1] + \text{imm}_{11:0}];$
Syntax:	<code>lw rd, imm_{11:0}(rs1)</code>
Encoding:	iiii iiiiiiii ssss s010 dddd d000 0011

Příklad 1

Uložme slovo z paměti na adrese hexadecimálně 0x4 do registru č.11:

`lw x11,0x4(x0)`

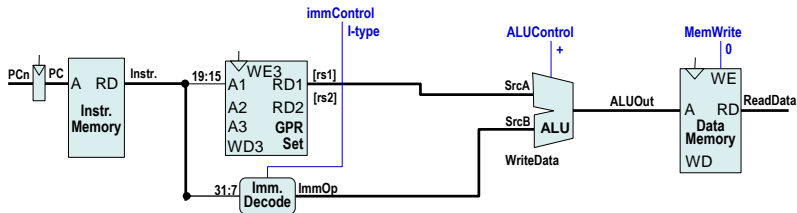
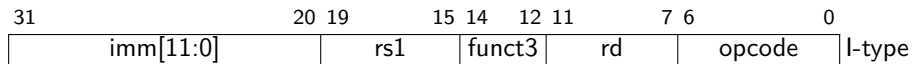


Závěr: Strojový kód instrukce `lw x11,0x4(x0)` je hexadecimálně 0x00402583.

Jednocykový procesor – návrh – realizace load word: `lw`

`lw rd, imm11:0(rs1)` $rd \leftarrow \text{Mem}[[rs1] + \text{imm}_{11:0}];$

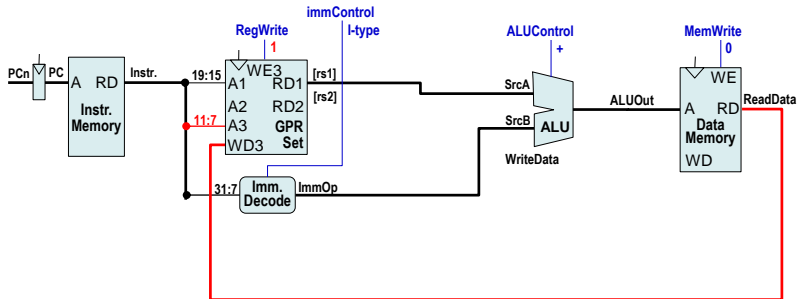
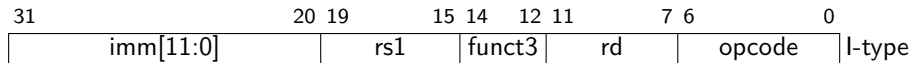
rs1 = básová adresa, **imm** = offset vůči básové adrese, **rd** = cílový registr



Jednocykový procesor – návrh – realizace load word: `lw`

`lw rd, imm11:0(rs1)` $rd \leftarrow \text{Mem}[[rs1] + \text{imm}_{11:0}];$

rs1 = bázeová adresa, **imm** = offset vůči bázeové adrese, **rd** = cílový registr

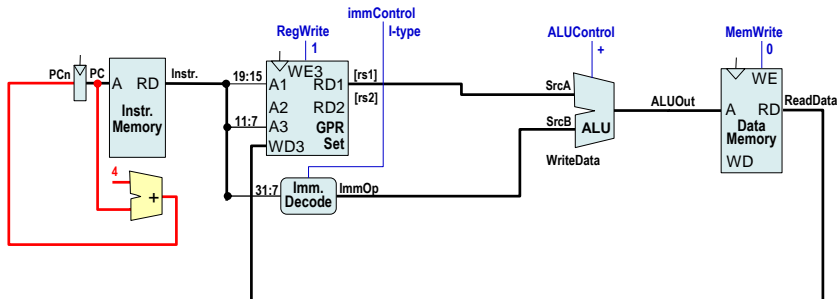
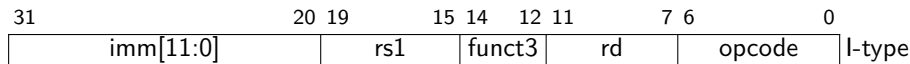


Jednocykový procesor – návrh – realizace load word: **lw**

lw rd, imm_{11:0}(rs1)

$rd \leftarrow \text{Mem}[[rs1] + \text{imm}_{11:0}];$

rs1 = básová adresa, **imm** = offset vůči básové adrese, **rd** = cílový registr

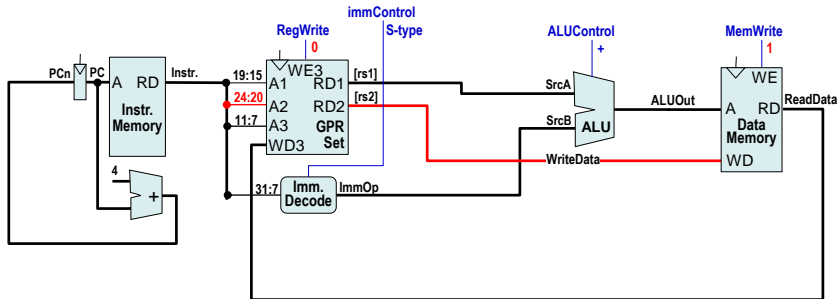
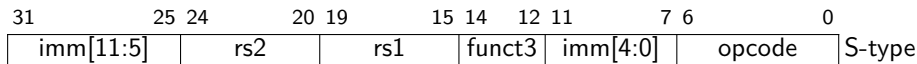


Jednocykový procesor – návrh – realizace store word: **sw**

sw **rs2**, **imm**_{11:0}(**rs1**)

$\text{Mem}[[\text{rs1}] + \text{imm}_{11:0}] \leftarrow [\text{rs2}];$

rs1 = básová adresa, **imm** = offset vůči básové adrese, **rs2** = zdrojový registr

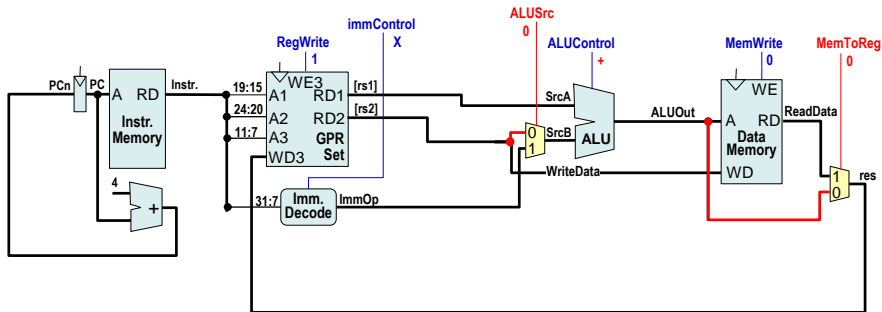
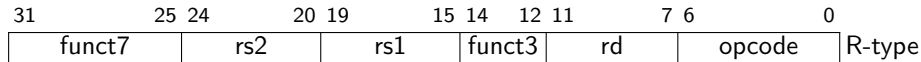


Jednocykový procesor – návrh – realizace součtu: add

add rd, rs1, rs2

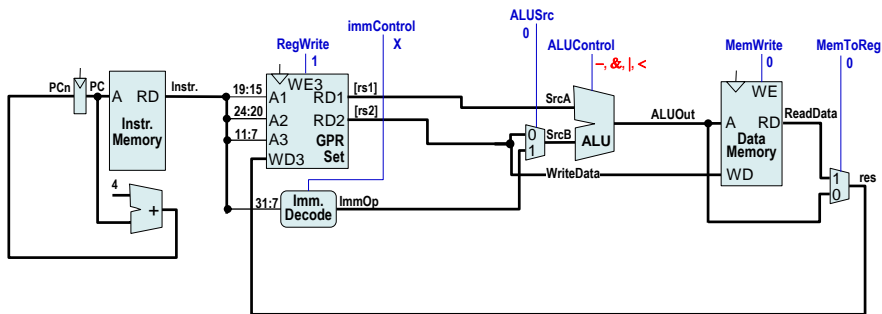
$rd \leftarrow [rs1] + [rs2];$

rs1, rs2 = zdroje, rd = cílový registr, funct7 = operace součtu



Jednocykový procesor – návrh – realizace instrukcí: **sub**, **and**, **or**, **slt**

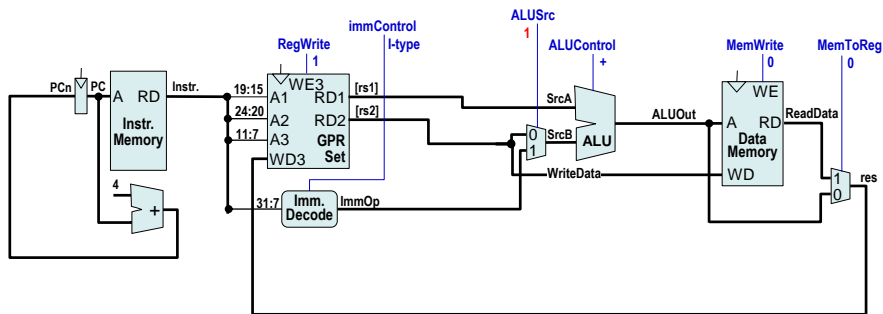
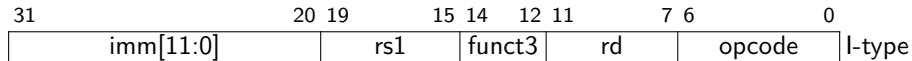
Jediné v čem se liší od **add** je operace ALU. Datapath je beze změny. Rozdíl je v hodnotě signálu v ALUControl. Počet bitů signálu ALUControl vyplývá z celkového počtu operací vykonávaných jednotkou ALU.



Jednocykový procesor – návrh – realizace součtu: `addi`

`addi rd, rs1, imm11:0` $rd \leftarrow [rs1] + imm_{11:0};$

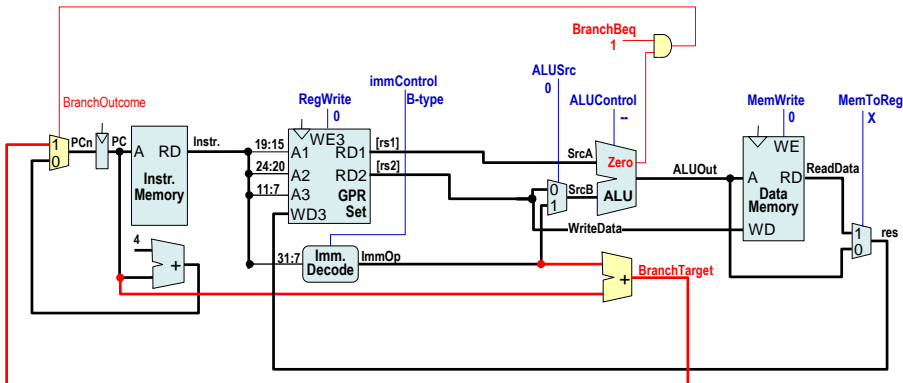
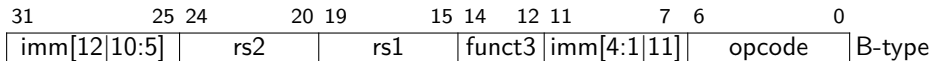
Instrukce `addi` patří do kategorie I – immediate, podobně jako `lw`. Datovou cestu tedy již máme. Rozdíl v řídicích signálech.



Jednocykový procesor – návrh – realizace beq

beq rs1, rs2, imm_{12:1}

if [rs1] == [rs2] go to [PC]+{imm_{12:1}, '0'};
else go to [PC]+4;

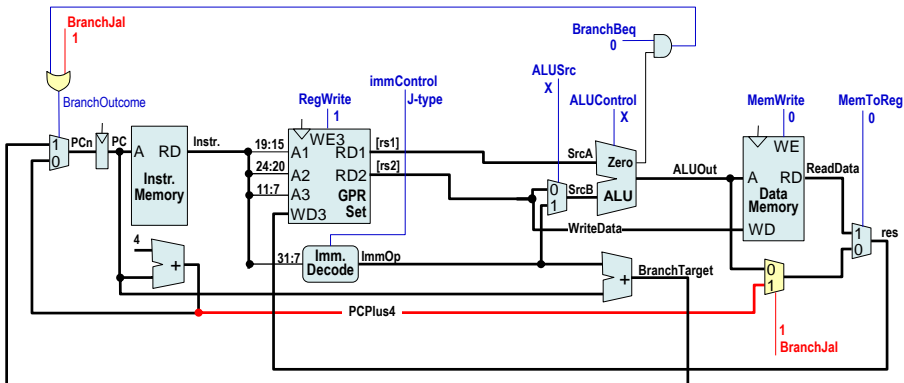


Jednocykový procesor – návrh – realizace jal

jal rd, imm_{20:1}

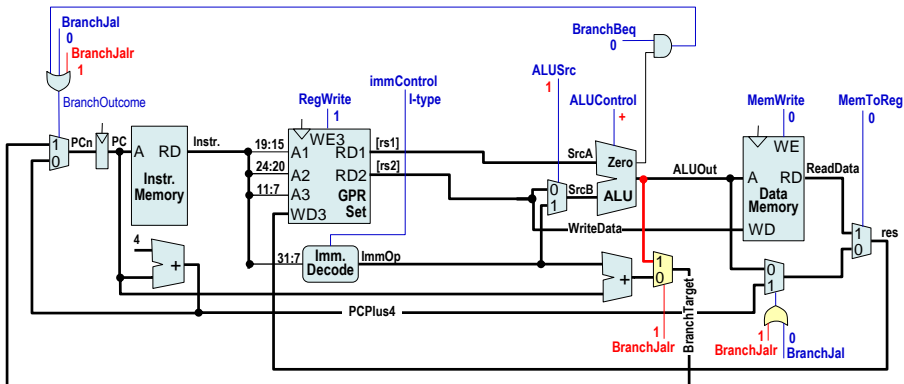
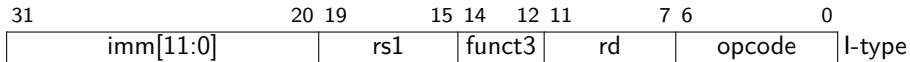
$rd \leftarrow [PC] + 4;$

$go\ to\ [PC] + \{imm_{20:1}, '0'\};$



Jednocyklový procesor – návrh – realizace jalr

```
jalr rd, rs1, imm11:0      rd ← [PC]+4;  
                             go to [rs1]+imm11:0;
```



Proč jsou `jal` a `jalr` tak důležité?

Instrukce `jal` a `jalr` v sobě zahrnují funkcionalitu instrukcí bezpodmíněného skoku: `j` (jump) a `jr` (jump register), a instrukce pro návrat z podprogramu: `ret` (return). Od `j`, `jr` a `ret` bychom požadovali:

Syntax	Semantics
<code>j imm20:1</code>	go to $[PC] + \{imm_{20:1}, '0'\}$;
<code>jr rs1</code>	go to $[rs1]$;
<code>ret</code>	go to $[x1]$;

Nicméně toho můžeme dosáhnout pomocí:

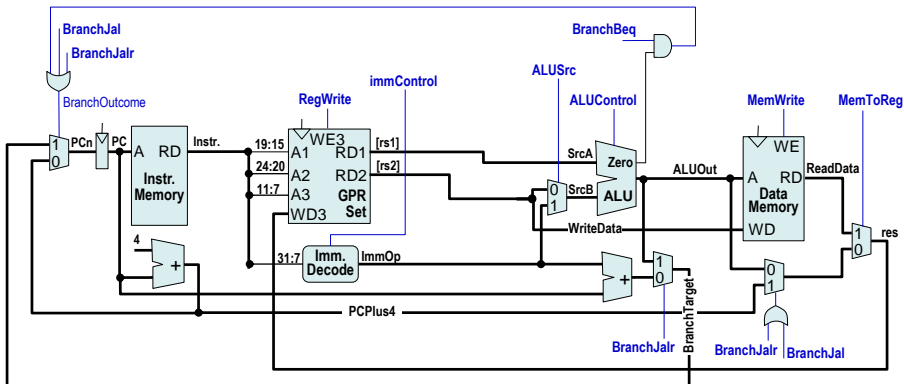
Syntax	Implementation	Semantics
<code>j imm20:1</code>	<code>jal x0,imm20:1</code>	$x0 \leftarrow [PC] + 4$; go to $[PC] + \{imm_{20:1}, '0'\}$;
<code>jr rs1</code>	<code>jalr x0,rs1,0</code>	$x0 \leftarrow [PC] + 4$; go to $[rs1] + 0$;
<code>ret</code>	<code>jalr x0,x1,0</code>	$x0 \leftarrow [PC] + 4$; go to $[x1] + 0$;

Proto není `j`, `jr` a `ret` potřeba implementovat v naší mikroarchitektuře.

Poznámka: Volání podprogramu dle RISC-V konvence je: `jal x1,imm20:1`.

Hotovo – navržený jednocyklový procesor

Jaká může být maximální frekvence tohoto procesoru? Tzn. je třeba vypočítat latenci signálu na nejdelší (kritické) cestě při provedení instrukce. Musíme uvažovat všechny implementované instrukce.

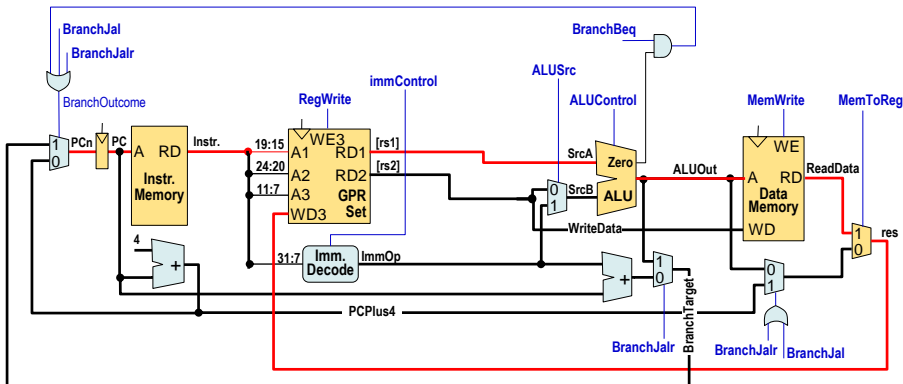


Jednocykový procesor – výkonnost:

$$IPS = IC / T_{CLK} = IPC * f_{CLK}$$

Zpoždění na kritické cestě (určuje ji nejdéle trvající instrukce) – instrukce **lw**:

$$T_{CLK} = t_{PC} + t_{Mem} + t_{GPRread} + t_{Mux} + t_{ALU} + t_{Mem} + t_{Mux} + t_{GPRsetup}$$



Jednocykový procesor – výkonnost:

$$IPS = IC / T_{CLK} = IPC * f_{CLK}$$

$$T_{CLK} = t_{PC} + t_{Mem} + t_{GPRread} + t_{ALU} + t_{Mem} + t_{Mux} + t_{GPRsetup}$$

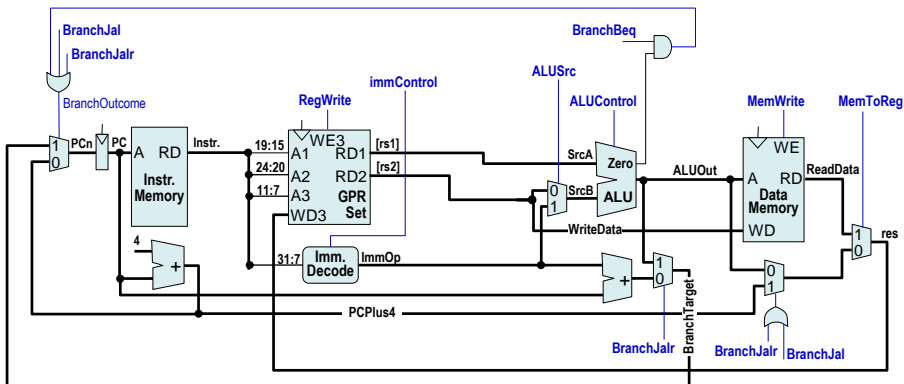
Předpokládejme:

- $t_{PC} = 0,3 \text{ ns}$
- $t_{Mem} = 20 \text{ ns}$
- $t_{GPRread} = 1,5 \text{ ns}$
- $t_{ALU} = 2 \text{ ns}$
- $t_{Mux} = 0,1 \text{ ns}$
- $t_{GPRsetup} = 0,1 \text{ ns}$

Pak $T_{CLK} = 44 \text{ ns} \rightarrow f_{CLKmax} = 22,7 \text{ MHz}$

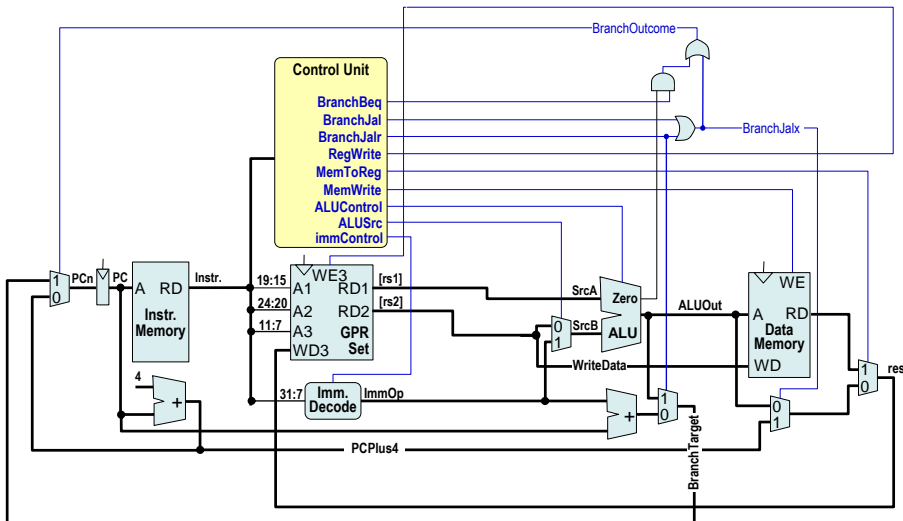
$$IPS = IPC * f_{CLK} = 22\,700\,000 \text{ instrukcí za sekundu} = 22,7 \text{ MIPS}$$

Návrh řadiče - rekapitulace stávajícího návrhu



Návrh řadiče

Funkcí řadiče je nastavit hodnoty jednotlivých řídicích signálů v závislosti od právě vykonávané instrukce.



Instrukce je určena především pomocí **opcode**:

opcode	Význam	Pro náš specifický případ
0110011	R-type (see funct7 and funct3)	add, sub, slt, or, and
0010011	ALU-imm (see funct3)	addi
0000011	Memory load (see funct3)	lw
0100011	Memory store (see funct3)	sw
1100011	Branch (see funct3)	beq
1101111	jal	jal
1100111	jalr	jalr

Pro instrukce typu R platí dále tabulka:

funct7	funct3	Instrukce
0000000	000	add
0100000	000	sub
0000000	010	slt
0000000	110	or
0000000	111	and

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20 10:1 11 19:12]										rd		opcode		J-type

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
imm[11:0]						rs1		010		rd		0000011	lw	
imm[11:5]			rs2			rs1		010	imm[4:0]			0100011	sw	
0000000			rs2			rs1		000	rd			0110011	add	
0100000			rs2			rs1		000	rd			0110011	sub	
0000000			rs2			rs1		010	rd			0110011	slt	
0000000			rs2			rs1		110	rd			0110011	or	
0000000			rs2			rs1		111	rd			0110011	and	
imm[11:0]						rs1		000	rd			0010011	addi	
imm[12 10:5]			rs2			rs1		000	imm[4:1 11]			1100011	beq	
imm[20 10:1 11 19:12]										rd			1101111	jal
imm[11:0]						rs1		000	rd			1100111	jalr	

Jednocykový procesor – návrh – řadič

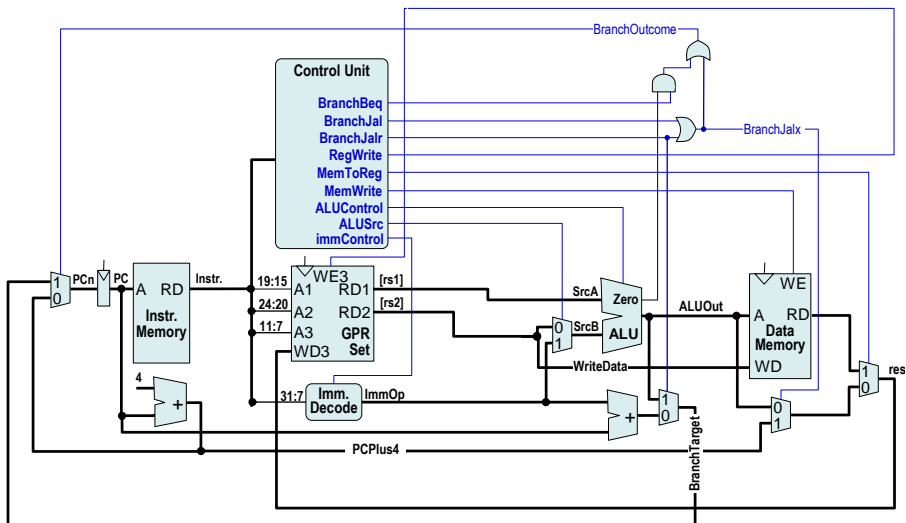
Hodnoty řídicích signálů jsou určeny pravdivostní tabulkou:

Instrukce	Opcode	Funct3	Funct7	ALUSrc	ALUControl	MemWrite	MemToReg	RegWrite	BranchBeq	BranchJal	BranchJalr	immControl
lw	0000011	010	don't care	Hodnoty řídicích signálů dle předchozích slajdů. ⁵								
sw	0100011	010	don't care									
add	0110011	000	0000000									
sub	0110011	000	0100000									
slt	0110011	010	0000000									
or	0110011	110	0000000									
and	0110011	111	0000000									
andi	0010011	000	don't care									
beq	1100011	000	don't care									
jal	1101111	don't care	don't care									
jalr	1100111	000	don't care									

Tudíž, řadič jednocykového CPU lze jednoduše realizovat jako **kombinační obvod**.

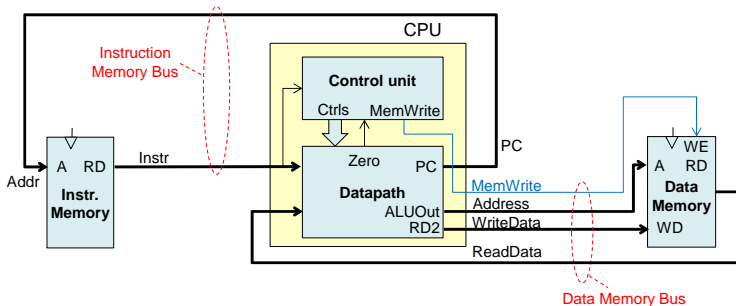
⁵Ponecháno na studentech.

Navržená mikroarchitektura jednocyklového procesoru



Jednocykový procesor – obecný pohled

- Hlavní úkol této přednášky byl návrh jednoduchého počítače tvořeného procesorem a oddělenými pamětmi instrukcí a dat.
- V našem návrhu procesor komunikuje s těmito pamětmi po oddělených sběrnicích.



- V obecném případě je sběrnice tvořena adresními, datovými a řídicími vodiči.
- V našem návrhu sběrnice pro komunikaci datovou pamětí jsou z důvodu jednoduchosti datové vodiče vedeny 2 krát (32 bitů pro WriteData, 32 bitů pro ReadData). Protože v našem případě nelze číst a zapisovat do datové paměti najednou, bylo by výhodnější použít obousměrné datové vodiče s řízením směru přenosu – viz BI-SAP přednáška č.10.



- Řadič je kombinační nebo sekvenční obvod:
 - ▶ vstupy: typ instrukce, stavové signály,
 - ▶ výstupy: řídicí signály.
- Funkce řadiče: V příslušný časový okamžik generovat řídicí signály a přijímat signály stavové. Poznámka pro náš specifický případ: náš řadič reaguje např. na stavový signál *Zero*.
- Řadič řídí činnost jednotlivých jednotek, koordinuje jejich aktivity a zajišťuje tok informace mezi nimi.
- Z hlavní paměti získává instrukce (sekvenci instrukcí), které mají být vykonány. Dekóduje je, a na základě typu instrukce nastaví příslušné hradla a datové cesty, aby mohla být instrukce vykonána.
- Obecně, **funkcí řadiče je generovat sekvenci řídicích signálů různým subsystémům počítače ve správném pořadí tak, aby byly vykonány požadované operace (aritmetické, změny toku programu aj.).**

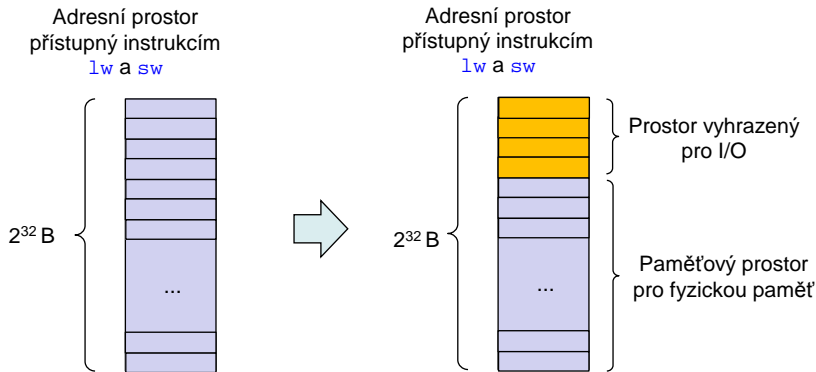


- Řadič **obvodový**:
 - ▶ Kombinační obvod (to byl náš případ),
 - ▶ Sekvenční obvod – stavový automat, apod.
- Řadič **mikroprogramovaný**⁶ (řízený mikroprogramem):
 - ▶ **Mikroprogram** je uložen v řídicí paměti řadiče a skládá se z **mikroinstrukcí**.
 - ▶ Mikroprogram implementuje programátorovi 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 zřetěžené procesory**, kdy každý stupeň vykonává jinou instrukci. Bylo by potřeba zajistit správné provedení všech instrukcí napříč stupni spolu s řešením hazardů.

⁶Víc o mikroprogramovaných řadičích v BI-JPO.

Komunikace s periferiemi (I/O)

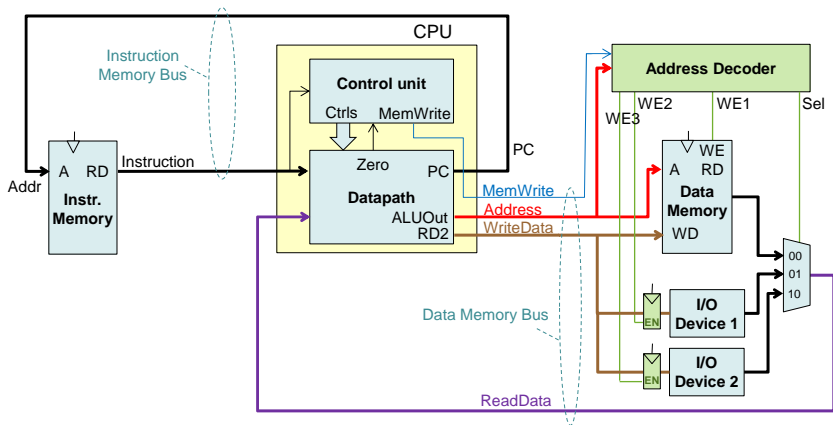
Idea: K tomu abychom komunistovali s vstupně/výstupními periferiemi (klávesnice, monitor, tiskárna) můžeme použít stejné rozhraní jako pro komunikaci s pamětí (instrukce `lw`, `sw`). Využijeme tedy **paměťově mapované I/O**.



Komunikace s periferiemi (I/O)

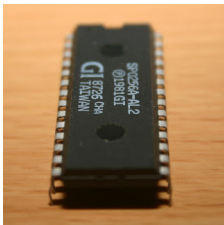
Idea: K tomu abychom komunikovali s vstupně/výstupními periferiemi (klávesnice, monitor, tiskárna) můžeme použít stejné rozhraní jako pro komunikaci s pamětí (instrukce **lw**, **sw**). Využijeme tedy **paměťově mapované I/O**.

- Adresový dekodér sleduje adresu na sběrnici *DataMemoryBus* a signál *MemWrite*. Pokud detekuje shodu s některou z I/O adres, patřičně zareaguje.

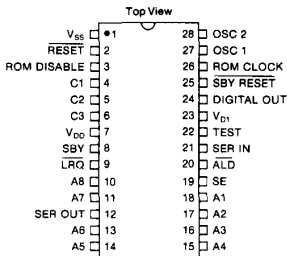


Příklad: Syntetizátor řeči I

- V angličtině se vyskytuje zhruba 60 různých alofonů (elementárních hlasových "zvuků"), ze kterých se skládají jednotlivá slova.
- Čip SP0256 je schopen generovat tyto jednotlivé zvuky.
- **Úkol:** Napišme ovladač pro tento čip.
 - ▶ Ovladač bude číst 5 položek (alofonů) od adresy 0x00000100 a ty pak pošle jednu za druhou do čipu SP0256.

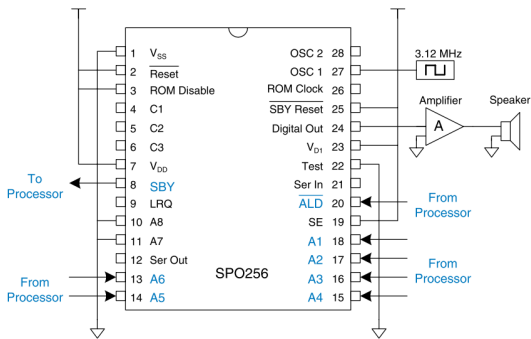


<http://little-scale.blogspot.com/2009/02/sp0256-a12-creative-commons-sample-pack.html>



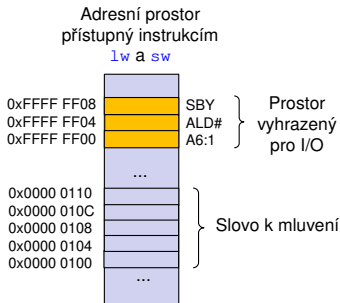
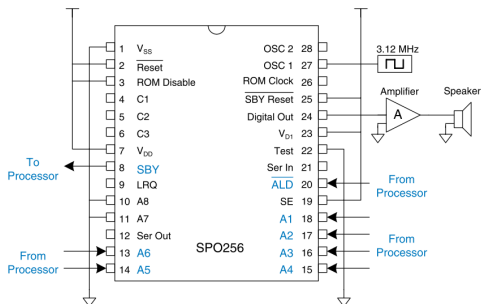
Příklad: Syntetizátor řeči II

- Procesor posílá 6-bitový alofon na piny A6:1.
- Vygenerovaný zvuk se objevuje na pinu *DigitalOut*.
- Piny *SBY* a \overline{ALD} slouží pro získání stavové informace a k ovládání čipu.
 - ▶ Pokud je výstup *SBY* roven 1, pak zařízení nemluví a čeká na příjem dalšího alofonu.
 - ▶ Na sestupné hraně vstupu \overline{ALD} pak přečte alofon (jeho hodnota je na vstupech A6:1) a začne mluvit.



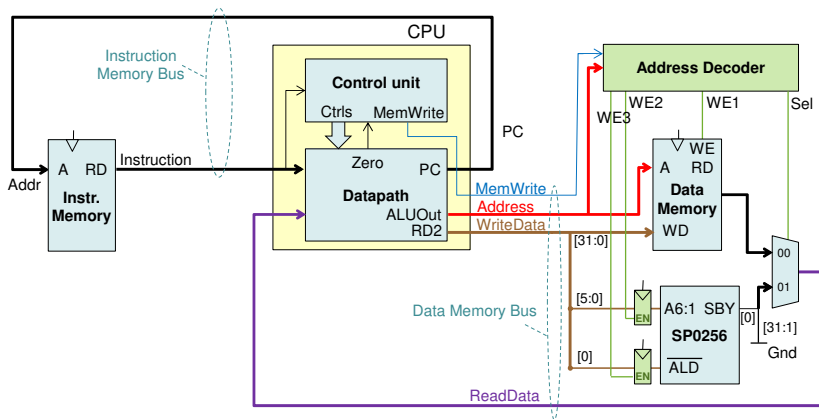
Příklad: Syntetizátor řeči III

- Zvolíme si mapování: Port A6:1 bude namapován do adresy 0xFFFF FF00, \overline{ALD} do adresy 0xFFFF FF04, a SBY do 0xFFFF FF08.



Příklad: Syntetizátor řeči IV

- Dolních 6 bitů *WriteData* se napojí na A6:1.
- Nejméně významný bit *WriteData* se přivede na \overline{ALD} .
- Podobně, hodnota SBY se čte z nejméně významného bitu *ReadData*.



Příklad: Syntetizátor řeči V – Ovladač

1. Nastav \overline{ALD} na 1.
2. Čekej dokud čip nenastaví SBY na 1 (indikace, že je připraven).
3. Zapiš 6-bitový alofon do A6:1.
4. Nastav \overline{ALD} na 0 (začni generovat zvuk).

```
init:
    addi x1,x0,1      // x1 = 1 (value to write to ALD#)
    addi x2,x0,20     // x2 = array size *4 (20 bytes)
    addi x3,x0,0x100  // x3 = array base address
    addi x4,x0,0      // x4 = 0 (array index)
start:
    sw x1,0xF04(x0)   // ALD#=1
loop:
    lw x5,0xF08(x0)   // x5 = SBY (monitor the state)
    beq x0,x5,loop    // loop until SBY == 0
    add x6,x3,x4      // x6 = address of allophone
    lw x7,0(x6)       // x7 = allophone
    sw x7,0xF00(x0)   // A6:1 = allophone
    sw x0,0xF04(x0)   // ALD# = 0 (to initiate speech)
    addi x4,x4,4      // increment array index
    beq x4,x2,done    // all allophones done?
    beq x0,x0,start   // repeat
done:
```

- Všimněte si, že procesor sleduje, zda je zařízení připraveno (sleduje hodnotu SBY). Tomu se říká **polling** (**aktivní vzorkování**). Bylo by však lepší nastavit na SBY **přerušování** a nechat procesor dělat i jinou činnost.



V zásadě existují 2 přístupy pro komunikaci mezi CPU a periferiemi:

- **Paměťově mapované I/O:**

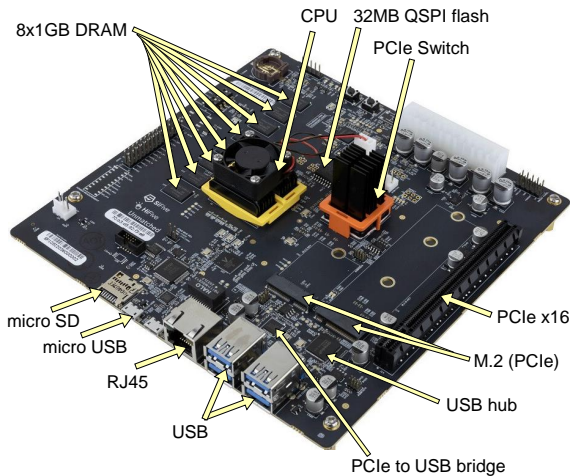
- ▶ Pro paměťově mapované I/O jsou části paměťového adresního prostoru vyhrazeny (přiřazeny) vstupně/výstupním zařízením. Čtení/záписy z/do těchto adres jsou interpretovány jako příkazy nebo přenosy dat z/do těchto zařízení. Paměťový systém ignoruje tyto operace (zná rozsah adres použitý pro I/O). Řadič I/O zařízení nicméně vidí tyto operace a reaguje na ně (ví které adresy jsou mu přiřazeny).
- ▶ V našem příkladu jsme použili **centralizovaný přístup**, kdy adresní dekodér řídil přístup ke všem I/O periferiím a paměti.
- ▶ V praxi se setkáme rovněž s **autonomním přístupem**, kdy každá I/O periferie má své speciální registry, ve kterých je uložena adresa tohoto zařízení. Tyto registry se inicializují při startu počítače. Periferie reaguje pouze, pokud detekuje shodu adres na sběrnici a jí přiřazené adresy.

- **Izolované (nebo také portově mapované) I/O:**

- ▶ Pro komunikaci s I/O se využívá separátní adresní I/O prostor.
- ▶ Jsou zapotřebí speciální I/O instrukce (například **in** a **out** pro x86), které do tohoto prostoru přistupují.

Rozdělení fyzického paměťového adresního prostoru I

Ukázka rozdělení fyzického paměťového adresního prostoru pro CPU, resp. SoC (System on Chip) FU740 obsahující 64-bitový 5-jádrový RISC-V procesor⁷:



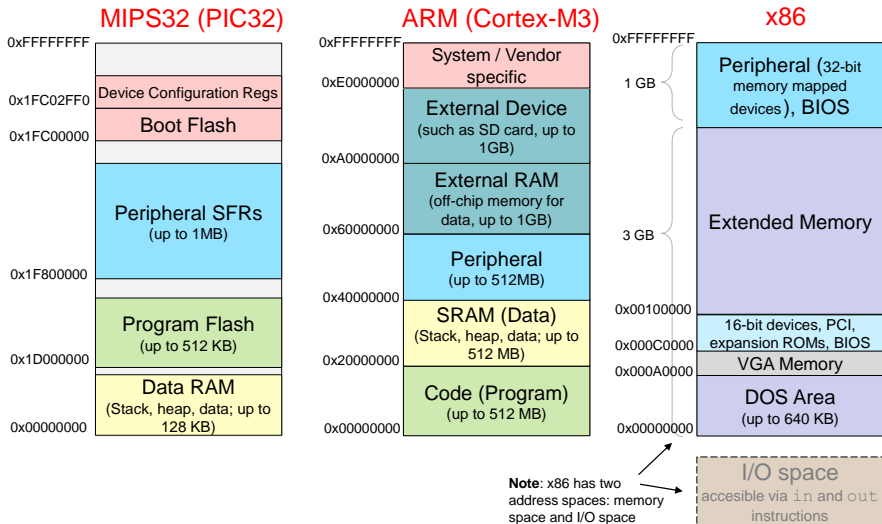
RISC-V (FU740 SoC)

0xFF FFFF FFFF	Peripheral (PCIe)
	DRAM (up to 64GB)
0x00 8000 0000	Peripheral (SPI)
0x00 2000 0000	Memory Controller, PCIe management
0x00 100B 0000	Peripheral (DMA, UART, I2C, QSPI, ...)
	Platform-Level Interrupt Controller (PLIC)
0x00 0C00 0000	L2 LIM (Loosely- Integrated Memory)
0x00 0800 0000	L2 cache controller
0x00 0000 0000	Core-Local Interruptor

⁷1xRV64IMAC, 4xRV64IMAFDC

Rozdělení fyzického paměťového adresního prostoru II

Rozdělení fyzického paměťového adresního prostoru se liší napříč architekturami a výrobci:



- ❶ John L. Hennessy, David A. Patterson: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 5th Edition. 2011.
- ❷ Paterson, D., Hennessy, V.: Computer Organization and Design, The HW/SW Interface. Elsevier, ISBN: 978-0-12-370606-5
- ❸ The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213", Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, December 2019.
- ❹ David Harris, Sarah Harris: Digital Design and Computer Architecture, 2nd Edition. Morgan Kaufmann.
- ❺ John Franco: What is Microprogramming and Why Should We Know About it?
<http://gauss.eecs.uc.edu/Courses/c4029/exams/Spring2013/Review/microcode.pdf>
- ❻ Cortex-M3 Technical Reference Manual:
<https://developer.arm.com/documentation/ddi0337/e/memory-map/about-the-memory-map>
- ❼ PIC32 Family Reference Manual: Section 3. Memory Organization.
<http://ww1.microchip.com/downloads/en/devicedoc/60001115h.pdf>
- ❽ SiFive U74 Core Complex Manual, 21G2.01.00.
- ❾ SiFive FU740-C000 Manual, v1p2.