MAM

3. Architektury procesorů

Stanislav Vítek
Katedra radioelektroniky
České vysoké učení technické v Praze

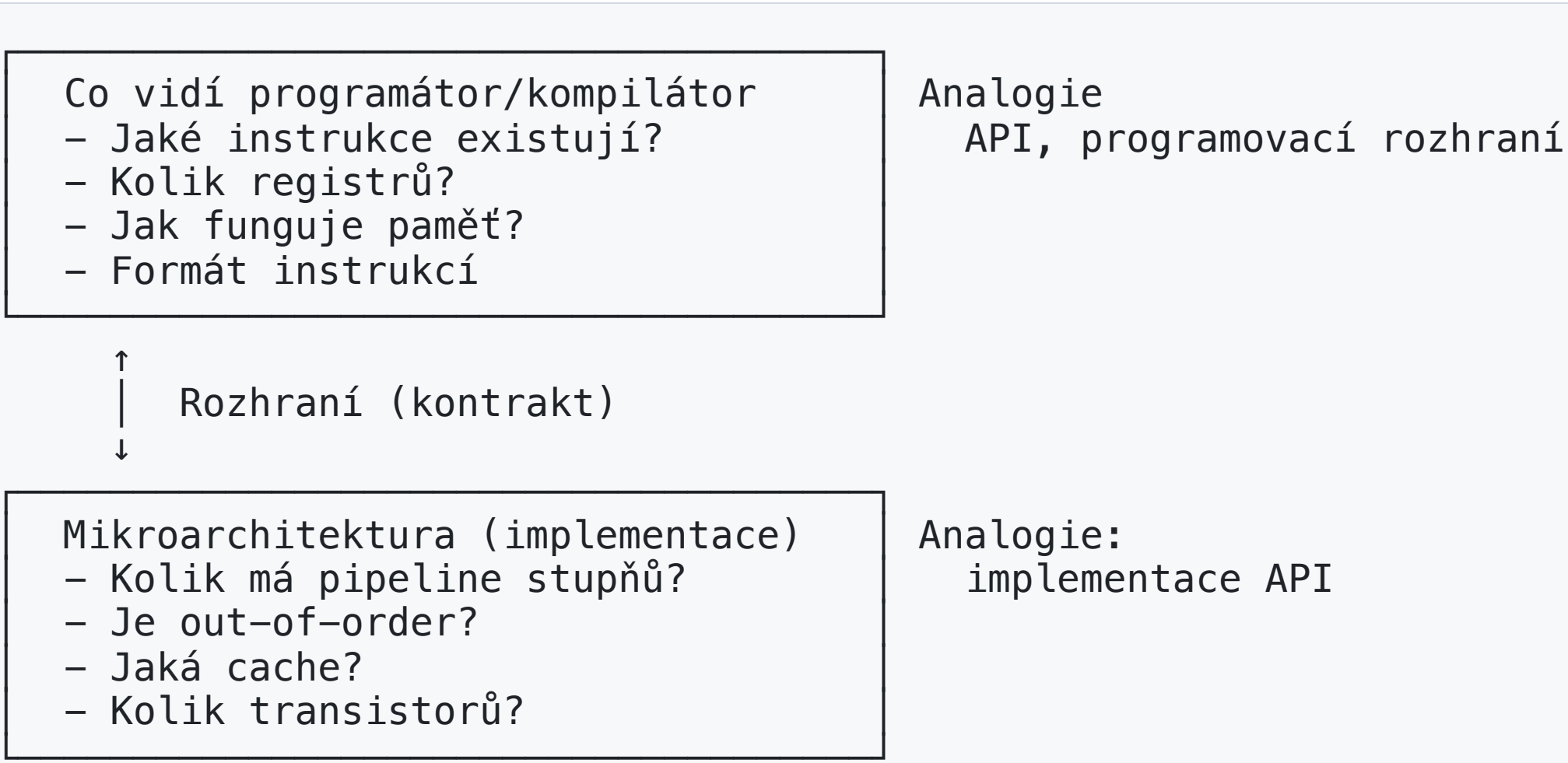
V přednášce 2 jsme viděli

- Přechod od jednoduchých obvodových řadičů (FSM) k mikrokontrolérům
- Základní komponenty: ALU, registry, řadič, paměť
- Jak procesor vykonává instrukce (fetch-decode-execute cyklus)
- Jednoduchou mikroarchitekturu s datovou cestou

V této přednášce

- Podíváme se na **konkrétní implementace** (ARM, RISC-V, x86)
- Zjistíme, že pod povrchem jsou si všechny procesory **velmi podobné**
- Hlavní rozdíl: **ISA** (Instruction Set Architecture) - "jazyk" procesoru
- Naučíme se **pipeline** - jak dosáhnout vyššího výkonu

ISA vs Mikroarchitektura



Příklad: x86 ISA, různé mikroarchitektury

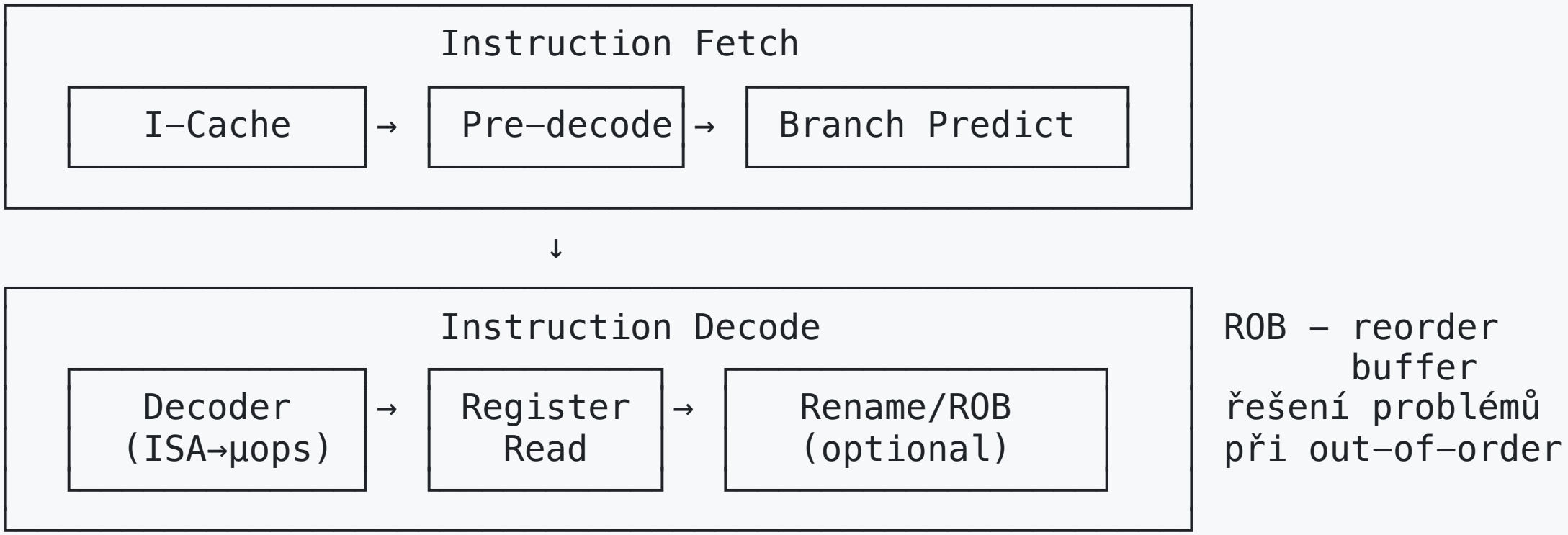
Stejná ISA (x86-64):

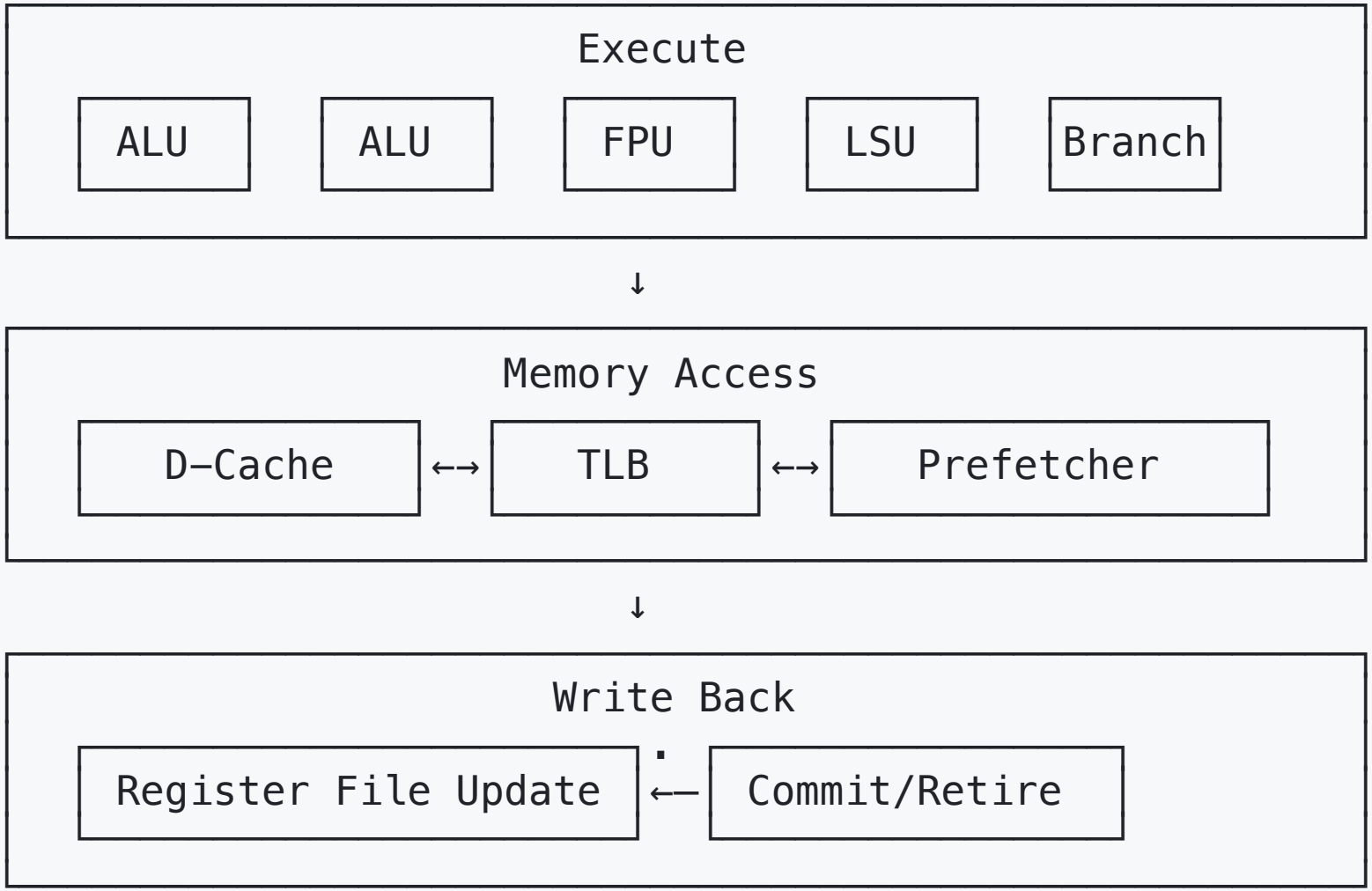
```
Intel Core i3 (2010)
├─ ISA: x86-64 ✓
├─ Pipeline: 14 stupňů
├─ Out-of-order: Ano
├─ Frekvence: 2.93 GHz
└─ Výkon: ~50 GFLOPS

Intel Core i9 (2023)
├─ ISA: x86-64 ✓ (stejná!)
├─ Pipeline: ~19 stupňů
├─ Out-of-order: Ano (lepší)
├─ Frekvence: 5.8 GHz
└─ Výkon: ~1000 GFLOPS
```

Na obou architekturách běží stejný software, ale implementace je zcela odlišná!

Společné stavební bloky - univerzální mikroarchitektura

Každý moderní procesor má:





FPU – floating point unit
LSU – load-store unit

TLB – translation lookaside unit
virt. -> fyzicka add.

Co je tedy odlišné? Hlavně ISA!

ISA definuje:

1. **Formát instrukcí:**

```
ARM:     32-bit pevná délka (nebo 16/32 Thumb-2)
x86:     1-15 bytů proměnná
RISC-V:  32-bit pevná (nebo 16/32 s RVC)
```

2. **Počet registrů:**

```
ARM:     16 GP registrů (ARMv7)
x86:     16 GP registrů (x86-64)
RISC-V:  32 GP registrů + x0=zero
```

3. **Instrukční sada:**

```
ARM:     ~200 base instrukcí + rozšíření
x86:     ~3000+ instrukcí (CISC)
RISC-V:  47 base (RV32I) + modulární rozšíření
```

4. **Sémantika (jak instrukce fungují):**

```
ARM:     Podmíněné vykonávání každé instrukce
x86:     Memory-to-memory operace
RISC-V:  Striktní load-store, x0=0
```

Proč je ISA důležitá?

1. Kompatibilita software:

```
Program zkompilovaný pro ARM
     ↓
Nemůže běžet na x86
     ↓
Musí se překompilovat (nebo emulovat)
```

2. Optimalizace kompilátoru:

- Kompilátor musí znát ISA
- Různé ISA → různé optimalizace
- RISC: kompilátor dělá více práce
- CISC: hardware dělá více práce

3. Výkonnostní charakteristiky:

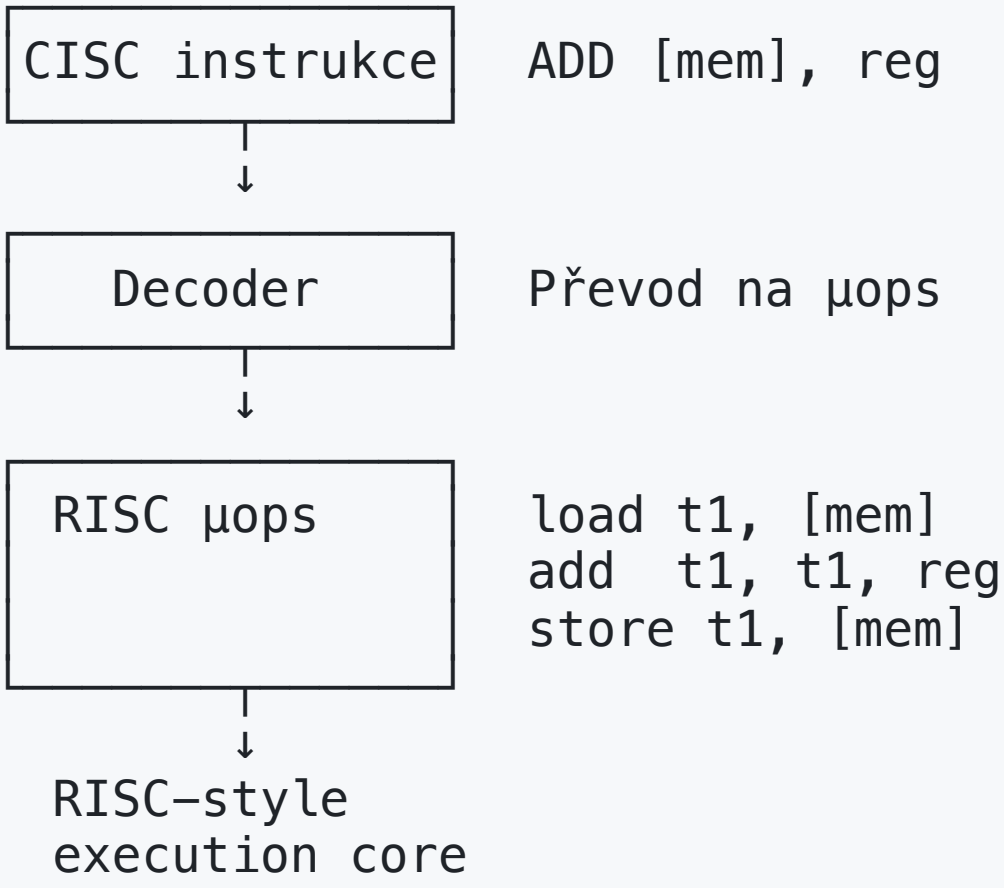
```
Jednoduchá ISA (RISC):
  + Jednodušší hardware
  + Nižší spotřeba
  + Vyšší frekvence možná
  - Více instrukcí na úlohu

Složitá ISA (CISC):
  + Méně instrukcí na úlohu
  + Hustší kód
  - Složitější hardware
  - Vyšší spotřeba
```

Konvergence mikroarchitektur

Zajímavé pozorování:

```
1990s:
  RISC procesory: jednoduché, in-order
  CISC procesory: složité, mikrokód

2025:
  RISC procesory: out-of-order, superskalární, složité
  CISC procesory: dekódují na RISC µops, pak stejné jako RISC

→ Mikroarchitektury konvergují!
→ ISA zůstává hlavní rozdíl
```

Moderní x86 interně:



O čem bude tato přednáška?

1. **Filozofie RISC vs CISC:**
   - Historické důvody, výhody/nevýhody
2. **ARM architektura:**
   - Konkrétní RISC implementace, instrukční sety (ARM/Thumb/Thumb-2)
3. **RISC-V architektura:**
   - Open-source alternativa, modulární design
4. **Pipeline (zřetězené zpracování):**
   - Společné pro všechny moderní procesory

Klíčový poznatek

> Mikroarchitektury procesorů jsou si velmi podobné. Hlavní rozdíl je v ISA - "jazyce", kterým s procesorem mluvíme. Výběr ISA ovlivňuje software kompatibilitu, ekosystém nástrojů, a do jisté míry i výkon a spotřebu.

1. Architektury RISC a CISC

CISC (1970-80s)

- Paměť byla **velmi drahá** → snaha minimalizovat velikost programu
- Kompilátory byly **primitivní** → složité instrukce usnadňovaly programování
- **Semantic gap**: snaha přiblížit strojový kód vysokoúrovňovým jazykům
- Mikroprogramování umožnilo snadnou implementaci složitých instrukcí

RISC (1980s)

- Výzkum ukázal, že 80% výkonu pochází z 20% instrukcí
- **Zlepšení kompilátorů**: dokážou efektivně optimalizovat jednoduchý instrukční set
- Paměť zlevnila → hustota kódu není tak kritická
- **Analýza**: složité instrukce jsou často pomalejší než sekvence jednoduchých
- Důraz na **rychlost vykonávání** místo hustoty kódu

1.2 Zásady RISC

1.2.1 Jednoduché instrukce s pevnou délkou

Pevná délka (32 bitů u většiny RISC):

Důsledky pevné délky:

- **Paralelní dekódování**: Víme kde jsou pole instrukce před dekódováním
- **Jednoduchý PC**: PC += 4 (vždy stejný přírůstek)
- **Predikce větvení**: Je snadné vypočítat cílovou adresu
- **Pipeline-friendly**: Každá fáze trvá podobně dlouho

Příklad dekódování:

```
Instrukce: 0x00A60333
           0000 0000 1010 0110 0000 0011 0011 0011

Okamžitě víme:
- Bity [6:0]   = opcode
- Bity [11:7]  = rd (destination register)
- Bity [19:15] = rs1 (source 1)
- Bity [24:20] = rs2 (source 2)
```

1.2.2 Architektura Load-Store

Striktní oddělení:

- **Pouze** LOAD/STORE instrukce přistupují k paměti
- **Všechny** aritmetické operace pouze registr-registr

```
# RISC přístup (3 instrukce):
LOAD  R1, [R2]      # Načti hodnotu z paměti
ADD   R3, R1, R4    # Aritmetika pouze mezi registry
STORE R3, [R5]      # Ulož výsledek do paměti

# CISC může (1 instrukce):
ADD [R5], [R2], R4  # Načti, sečti, ulož – vše najednou
```

Výhody Load-Store:

- **Uniformita**: Všechny ALU instrukce mají stejný formát
- **Predikovatelná latence**: Aritmetika = 1 cyklus, load = 2-3 cykly
- **Jednodušší pipeline**: Jasné rozdělení fází (MEM fáze jen pro L/S)
- **Efektivní forwarding**: Datové cesty jsou jednoduché

Nevýhody:

- Více instrukcí pro stejnou operaci
- Větší tlak na registry
- Potřeba více kódu (ale rychlejší vykonávání)

1.2.3 Velký registrový soubor

Typicky 32 registrů (RISC-V, MIPS) vs 8-16 (x86):

Proč více registrů?

- **Redukce přístupů do paměti**: paměť je 100-1000× pomalejší
- **Optimalizace kompilace**: více proměnných v registrech
- **Volání procedur/funkcí**: předávání parametrů bez stacku

1.2.5 Compiler vs Hardware trade-off

RISC řeší **v compile-time**:

- Plánování instrukcí (instruction scheduling)
- Alokace registrů
- Loop unrolling, software pipelining, dead code elimination

Příklad optimalizace:

```
# Před optimalizací (datový hazard):
ADD R1, R2, R3
SUB R4, R1, R5    # Stall! čeká na R1

# Po optimalizaci (instruction scheduling):
ADD R1, R2, R3
MUL R6, R7, R8    # Nezávislá instrukce sem
SUB R4, R1, R5    # Už je R1 ready
```

1.3 Zásady CISC

1.3.1 Proměnná délka instrukcí

Příklad x86 (1-15B):

```
NOP                          # 1B:  90
MOV AL, 42                   # 2B: B0 2A
ADD EAX, [EBX+4]             # 3B: 03 43 04
MOV [ESI+disp32], imm32      # 10+ B
```

Struktura x86 instrukce (detailní):

Prefix 0–4B	Opcode 1–3B	ModR/M 0–1B	SIB 0–1B	Disp. 0–4B	Immediate 0–4B	...

Detailní rozbor polí:

1. Prefix (0-4 byty) - Volitelné modifikátory:

```
Legacy prefixes (může být více současně):
- 0xF0: LOCK (atomická operace)
- 0xF2: REPNE/REPNZ (repeat while not zero)
- 0xF3: REP/REPE/REPZ (repeat)
- 0x2E, 0x36, 0x3E, 0x26, 0x64, 0x65: Segment overrides
- 0x66: Operand size override (16↔32 bit)
- 0x67: Address size override

REX prefix (x86-64 only, 1 byte):
```

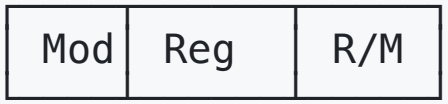
0100	W	R	X	B

```
  - W: 64-bit operand
  - R: Extension of ModR/M reg field
  - X: Extension of SIB index field
  - B: Extension of ModR/M r/m field
```

2. Opcode (1-3 byty) - Operační kód:

```
Určuje základní operaci:

1-byte opcode:
  0x01: ADD r/m32, r32
  0x03: ADD r32, r/m32
  0x05: ADD EAX, imm32
  0x50-0x57: PUSH reg
  0x90: NOP
  ...

2-byte opcode (escape s 0x0F):
  0x0F 0x84: JE/JZ (near jump)
  0x0F 0xAF: IMUL
  0x0F 0xB6: MOVZX (zero extend)
  ...

3-byte opcode (0x0F + další escape):
  0x0F 0x38 0xXX: SSE4, AVX instrukce
```

3. ModR/M byte (0-1 byte) - Adresování:




```
- Mod [7:6]: Addressing mode
  00: [reg] (indirect, no displacement)
  01: [reg + disp8] (8-bit displacement)
  10: [reg + disp32] (32-bit displacement)
  11: registr (direct, no memory)

- Reg [5:3]: První registr (nebo opcode extension)
  000: EAX/AX/AL/R8
  001: ECX/CX/CL/R9
  010: EDX/DX/DL/R10
  011: EBX/BX/BL/R11
  100: ESP/SP/AH/R12 (nebo SIB signál)
  101: EBP/BP/CH/R13
  110: ESI/SI/DH/R14
  111: EDI/DI/BH/R15

- R/M [2:0]: Druhý registr nebo memory
  (význam závisí na Mod)
```

4. SIB byte (0-1 byte) - Scale-Index-Base:

```
Použije se když ModR/M.R/M = 100 a Mod != 11


┌─────────┬─────────┬─────────┐
│ Scale   │ Index   │  Base   │
└─────────┴─────────┴─────────┘
 2bit     3bit      3bit

Výpočet adresy:
  EA = Base + Index×(2^Scale) + Displacement

- Scale [7:6]: Škálovací faktor
  00: ×1
  01: ×2
  10: ×4
  11: ×8

- Index [5:3]: Index registr
  (ESP=100 značí "bez indexu")

- Base [2:0]: Base registr
  (EBP=101 má speciální význam)

Příklad: [EAX + ECX×4 + 8]
  Base=EAX, Index=ECX, Scale=2 (×4), Disp=8
```

5. Displacement (0/1/2/4 byty) - Offset:

```
Přičítá se k vypočtené adrese
- 0 bytů: Mod=00, Base!=EBP
- 1 byte:  Mod=01 (disp8, -128 až +127)
- 4 byty:  Mod=10 (disp32, -2G až +2G)
```

6. Immediate (0/1/2/4 byty) - Konstanta:

```
Hodnota přímo v instrukci
Velikost závisí na operandu size a typu instrukce
```

Příklad kompletního x86 kódování:

```
ADD DWORD PTR [EBX + ECX×4 + 0x10], 0x42

Binárně:
```

Opcode	ModR/M	SIB	Disp	Imm
81	44	8B	10	42
1 byte	1 byte	1 byte	1 byte	4 bytes

```
Dekódování:
1. Opcode = 0x81: ADD r/m32, imm32
2. ModR/M = 0x44:
   - Mod = 01 (disp8)
   - Reg = 000 (opcode extension pro ADD)
   - R/M = 100 (SIB následuje)
3. SIB = 0x8B:
   - Scale = 10 (×4)
   - Index = 001 (ECX)
   - Base = 011 (EBX)
4. Disp = 0x10 (16 v decimálním)
5. Imm = 0x00000042 (66 v decimálním)

Výsledek: [EBX + ECX×4 + 16] += 66

Celkem: 8 bytů
```

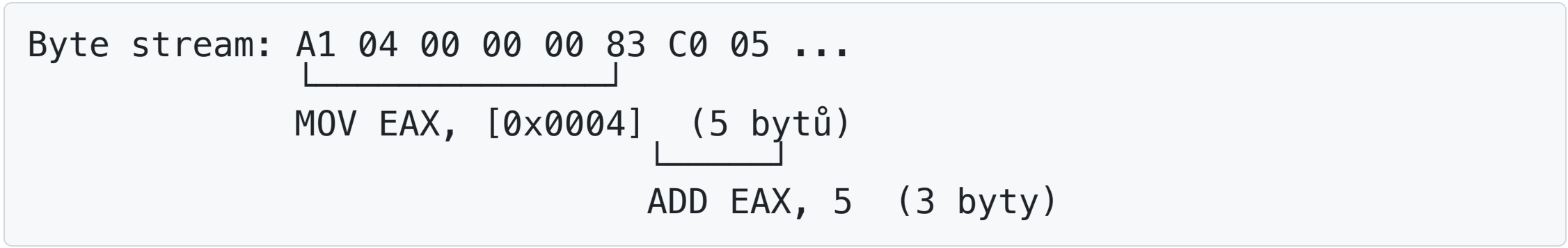
Proč je x86 složité?

```
Důsledky proměnné délky:
1. Sekvenční dekódování (nemůžeme paralelně)
2. Branch target alignment komplikace
3. Předekódování nutné (µop cache)
4. Složitý front-end

Výhoda:
- Hustší kód (menší programy)
- Kompatibilita zpět (16/32/64 bit)
```

Důsledky:

- **Složité dekódování**: Musíme sekvenčně číst, nevíme kde instrukce končí
- **Předekódování**: x86 přidává L1 I-cache s předekódovanými značkami
- **CISC → µops**: Interní převod na RISC-like mikrooperace
- **Alignment nightmare**: Instrukce může zasahovat více cache-lines

Dekódování:


Byte stream: A1 04 00 00 00 83 C0 05 ...
MOV EAX, [0x0004]
(5 bytů)
ADD EAX, 5
(3 byty)

1.3.2 Bohaté adresovací režimy

x86 podporuje:

```
MOV EAX, [EBX]                  # Nepřímé
MOV EAX, [EBX + 4]              # S offsetem
MOV EAX, [EBX + ECX]            # Indexované
MOV EAX, [EBX + ECX*4]          # Škálované
MOV EAX, [EBX + ECX*4 + 8]      # Plné: base+index*scale+disp
```

Výpočet efektivní adresy:

```
EA = Base + Index×Scale + Displacement
EA = EBX + ECX×4 + 8

V hardwaru: speciální jednotka AGU (Address Generation Unit)
```

Důsledky:

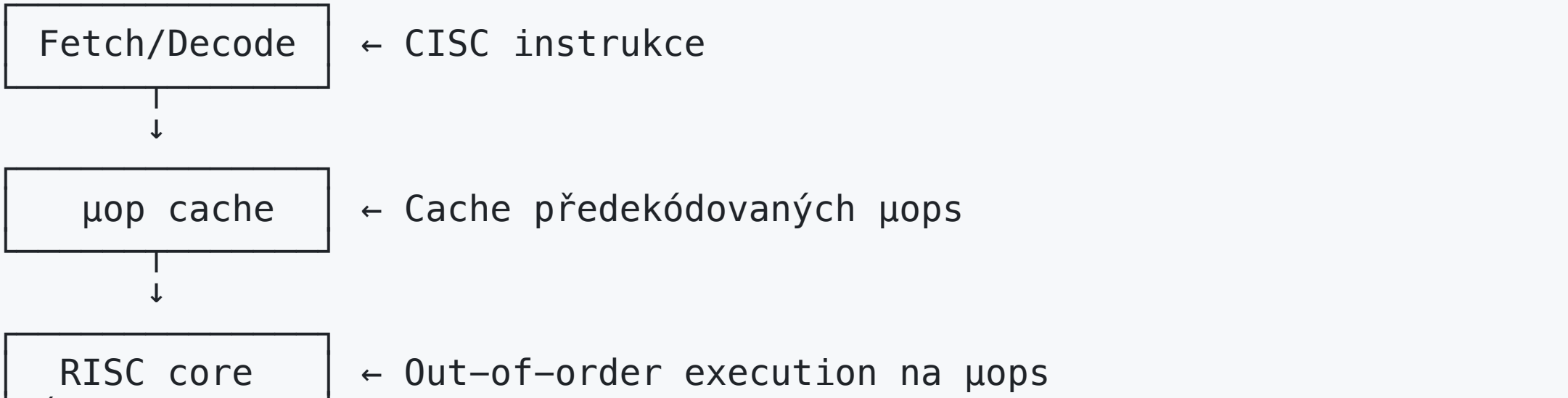
- **Jeden load = komplexní výpočet**: víc než jen sčítání
- **Latence**: složitější adresování = delší latence
- **Hustota kódu**: ušetří instrukce, ale pomalejší

1.3.3 Mikrokód a mikroarchitektura

CISC instrukce → mikrooperace (µops):

```
CISC instrukce: PUSH EAX
                    ↓
µop1: SUB ESP, 4      # Dekrementuj stack pointer
µop2: STORE EAX, [ESP] # Ulož hodnotu
```

Mikroarchitektura moderního x86:



1.3.4 Paměťové operace v instrukcích

CISC dovoluje:

```
ADD [memory], register    # Read-Modify-Write v jedné instrukci
INC [counter]             # Inkrementuj přímo v paměti
XCHG [mem1], [mem2]       # Swap přímo v paměti
```

Problém pro pipeline:

```
ADD [EBX], EAX:
  Cyklus 1: Fetch instrukce
  Cyklus 2: Dekóduj
  Cyklus 3: Load [EBX]
  Cyklus 4: ADD
  Cyklus 5: Store result → [EBX]

Celkem: 5 cyklů pro jednu instrukci!
```

RISC ekvivalent (3 instrukce, ale rychleji s pipelining):

```
LOAD  R1, [R2]    # IF ID EX MEM WB
ADD   R1, R1, R3  #    IF ID EX  MEM WB
STORE R1, [R2]    #       IF ID  EX  MEM WB
                  # 3 instrukce, ale 5 cyklů s pipelining
```

1.3.5 Méně registrů, víc stacku

x86-32 má pouze 8 GP registrů:

- EAX, EBX, ECX, EDX (ale často special-purpose)
- ESI, EDI (index)
- EBP (base pointer)
- ESP (stack pointer)

Důsledky:

- **Register pressure**: kompilátor často musí spillovat do paměti
- **Časté load/store**: i když má x86 memory operace, je to pomalé
- **Stack-heavy**: parametry funkcí často přes stack

Srovnání:

```
int sum(int a, int b, int c, int d, int e) {
    return a + b + c + d + e;
}

// x86: parametry 3-5 jdou přes stack (málo registrů)
// ARM/RISC-V: parametry 1-8 v registrech (dost registrů)
```

1.4 Konvergence architektur

"RISC vs CISC" se rozmazává:

Moderní x86:

- Interně RISC (µops)
- Out-of-order execution
- Register renaming (96+ fyzických registrů interně!)
- Složitý front-end, ale efektivní core

Moderní ARM:

- Přidává složitější instrukce (NEON SIMD)
- Podmíněné vykonávání = typ CISC vlastnosti
- Násobné load/store (LDM/STM)

Závěr:

- **ISA** (co vidí programátor): RISC = jednoduchý, CISC = složitý
- **Mikroarchitektura** (jak to běží): všichni používají RISC principy
- **Výkon**: záleží na implementaci, ne na ISA
- **Energie**: RISC stále vítězí (méně transistorů na dekódování)

2. Architektura ARM

2.1 Historie a vývoj rodiny ARM

2.1.1 Původ a filozofie (1983-1990)

Acorn RISC Machine (1983):

- Acorn Computers (UK), inspirace výzkumem Berkeley RISC a Stanford MIPS
- Cíl: jednoduchý, energeticky efektivní procesor pro Acorn počítače
- **ARM1 (1985)**: První prototyp, 25,000 tranzistorů
- **ARM2 (1987)**: První komerční, použitý v Acorn Archimedes

Klíčová designová rozhodnutí:

- 32-bit od začátku (většina konkurence byla 8/16-bit)
- Podmíněné vykonávání každé instrukce
- Load/Store architektura s 27 instrukcemi
- Žádná cache (šetření tranzistorů)

2.1.2 Klasické rodiny ARM (1990-2004)

ARM7 (1994):

3stupňové pipeline

- **3stupňové pipeline**: Fetch, Decode, Execute
- Von Neumannova architektura (jedna sběrnice)
- ARM7TDMI: T=Thumb, D=Debug, M=Multiplier, I=ICE
- **Použití**: Gameboy Advance, Nokia telefony, iPod
- **Výkon**: 0.9 MIPS/MHz, ~50 MHz
- **Spotřeba**: ~0.06 mW/MHz

ARM9 (1998):

5stupňové pipeline

- **5stupňové pipeline**: klasický RISC design
- Harvardská architektura (oddělené I/D sběrnice)
- Memory Management Unit (MMU) → podpora OS
- **Výkon**: 1.1 MIPS/MHz, až 300 MHz
- **Použití**: Nintendo DS, feature phones

ARM11 (2002):

- **8stupňové pipeline**: vyšší frekvence (až 1 GHz)
- SIMD rozšíření (ARMv6)
- Cache: L1 + volitelné L2
- **Použití**: Raspberry Pi 1, iPhone 1/3G
- Začátek éry high-performance ARM

2.1.3 Cortex éra (2004-současnost)

Rozdělení do profilů:

Cortex-M (Microcontroller):

- Ultra-nízká spotřeba, real-time
- Thumb-2 pouze (žádný ARM režim)
- NVIC (Nested Vectored Interrupt Controller)
- Determinističnost: přerušení za 12 cyklů
- **M0/M0+**: 0.9 µW/MHz, 32 KB flash stačí
- **M3**: Mainstream MCU (STM32F1/F2)
- **M4**: DSP instrukce, FPU (STM32F4)
- **M7**: 2000 CoreMark, double-precision FPU
- **M33/M55**: TrustZone, Helium (MVE)

Cortex-R (Real-time):

- Safety-critical aplikace
- Deterministické cache, ECC paměti
- Dual-core lockstep režim
- **Použití**: Airbags, ABS, hard-disky

Cortex-A (Application):

- High-performance, Linux/Android
- Out-of-order execution (od A9)
- Multi-core, big.LITTLE
- **A53**: Efektivní (1.5 GHz, 2.3 DMIPS/MHz)
- **A72**: Výkonný (2.5 GHz, 4.7 DMIPS/MHz)
- **A78/X1**: Moderní high-end

2.1.4 ARMv8/v9 - 64-bit éra (2011-)

ARMv8-A (2011):

- **AArch64**: nový 64-bit režim
- **AArch32**: kompatibilita s ARMv7
- 31 × 64-bit registrů (x0-x30)
- Odstraněno podmíněné vykonávání (mimo flag set)
- NEON standard (dříve volitelné)

ARMv9 (2021):

- Scalable Vector Extension 2 (SVE2)
- Confidential Compute Architecture (CCA)
- Nárůst výkonu: +30% IPC za generaci

2. Architektura ARM

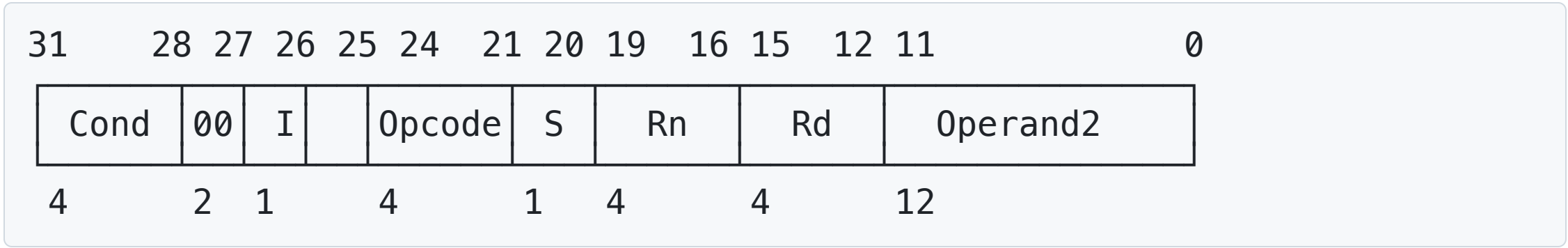
2.2 Instrukční sady

2.2.1 ARM 32-bitový režim (původní)

Základní terminologie:

- **rd** (destination register): Cílový registr, kam se zapíše výsledek
- **rs** / **rn** (source register): Zdrojový registr s operandem
- **rm**: Další zdrojový registr (často druhý operand)
- **opcode**: Operační kód - určuje typ operace
- **funct**: Function - doplňující specifikace operace
- **imm** / **immediate**: Konstanta zakódovaná přímo v instrukci

Formát ARM instrukce - Data Processing:



Následuje detailní popis polí.

1. Cond [31:28] - Condition code (4 bity)

```
Určuje podmínku pro vykonání instrukce:

0000 (0x0) - EQ:  Equal                (Z=1)
0001 (0x1) - NE:  Not Equal            (Z=0)
0010 (0x2) - CS:  Carry Set            (C=1)
0011 (0x3) - CC:  Carry Clear          (C=0)
0100 (0x4) - MI:  Minus/Negative       (N=1)
0101 (0x5) - PL:  Plus/Positive        (N=0)
0110 (0x6) - VS:  Overflow Set         (V=1)
0111 (0x7) - VC:  Overflow Clear       (V=0)
1000 (0x8) - HI:  Higher (unsigned)    (C=1 && Z=0)
1001 (0x9) - LS:  Lower/Same           (C=0 || Z=1)
1010 (0xA) - GE:  Greater/Equal        (N=V)
1011 (0xB) - LT:  Less Than            (N!=V)
1100 (0xC) - GT:  Greater Than         (Z=0 && N=V)
1101 (0xD) - LE:  Less/Equal           (Z=1 || N!=V)
1110 (0xE) - AL:  Always (unconditional)
1111 (0xF) - NV:  Never (deprecated)

Příklad: ADDGT = ADD pokud Greater Than
         Cond=1100 (GT)
```

2. Type [27:26] + I [25] - Typ instrukce (3 bity)

```
00 = Data processing (ALU operace)
     I bit:
       0 = Operand2 je registr (možná se shiftem)
       1 = Operand2 je immediate

01 = Load/Store
     I bit určuje offset typ

10 = Branch
11 = Coprocessor
```

3. Opcode [24:21] - Operace (4 bity)

```
Pro data processing:
0000 – AND:  Rd = Rn AND Operand2
0001 – EOR:  Rd = Rn XOR Operand2
0010 – SUB:  Rd = Rn – Operand2
0011 – RSB:  Rd = Operand2 – Rn (Reverse SUB)
0100 – ADD:  Rd = Rn + Operand2
0101 – ADC:  Rd = Rn + Operand2 + Carry
0110 – SBC:  Rd = Rn – Operand2 – !Carry
0111 – RSC:  Rd = Operand2 – Rn – !Carry
1000 – TST:  Nastaví flags podle Rn AND Operand2 (no write)
1001 – TEQ:  Nastaví flags podle Rn XOR Operand2
1010 – CMP:  Nastaví flags podle Rn – Operand2
1011 – CMN:  Nastaví flags podle Rn + Operand2
1100 – ORR:  Rd = Rn OR Operand2
1101 – MOV:  Rd = Operand2 (Rn ignored)
1110 – BIC:  Rd = Rn AND NOT Operand2 (Bit Clear)
1111 – MVN:  Rd = NOT Operand2 (Move Negative)
```

4. S [20] - Set condition codes (1 bit)

```
0 = Neaktualizuj flags (CPSR zůstane)
1 = Aktualizuj N, Z, C, V flags v CPSR

Příklad:
  ADD  R0, R1, R2    # S=0: R0 = R1 + R2, flags nezmění
  ADDS R0, R1, R2    # S=1: R0 = R1 + R2, nastaví flags
```

5. Rn [19:16] - První operand registr (4 bity)

```
Registr s první hodnotou pro operaci
4 bity → 16 možných registrů (R0–R15)

Výjimka: U MOV/MVN je Rn ignorován (nebere se v potaz)
```

6. Rd [15:12] - Destination registr (4 bity)

```
Cílový registr, kam se zapíše výsledek
4 bity → 16 možných registrů (R0–R15)

Speciální případy:
- Rd = R15 (PC): Zápis do PC = branch
- U TST/TEQ/CMP/CMN: Rd se nepoužívá (pouze flags)
```

7. Operand2 [11:0] - Druhý operand (12 bitů)

```
Dva režimy podle I bitu:

A) I=0: Registr s volitelným shiftem (12 bitů)
```

Shf	Rm	0/1	Typ	Amount
1	4 bit	1	2bit	5bit

```
- Rm [3:0]: Registr s hodnotou
- Type [6:5]: LSL/LSR/ASR/ROR
- Amount: buď 5-bit konstanta [11:7], nebo registr Rs[11:8]
```

Příklad: R2, LSL #3 = Shift R2 left o 3 bity

B) I=1: Immediate hodnota (12 bitů)

Hodnota = Immed rotovaná right o (2×Rotate) bitů

Příklad:
  Immed=0xFF, Rotate=0 → hodnota = 0x000000FF
  Immed=0xFF, Rotate=1 → hodnota = 0xC000003F (rotace o 2)
  Immed=0xFF, Rotate=8 → hodnota = 0xFF000000 (rotace o 16)

Proč rotace? Umožňuje reprezentovat více hodnot s 8 bity!

Příklad kompletního kódování:

```
ADDS R3, R4, R5, LSL #2

Rozpis:
- Instrukce: ADD (opcode=0100)
- S=1 (ADDS - set flags)
- Rn=R4 (0100)
- Rd=R3 (0011)
- Rm=R5 (0101), LSL, shift amount=2
- Cond=AL (1110)
- I=0 (registr operand)

Binárně:
31  28 27 26 25 24  21 20 19  16 15  12 11  7 6 5 4  3 2  0
┌──────┬──┬──┬──┬──────┬──┬──────┬──────┬─────┬──┬──┬──┬──────┐
│ 1110 │00│ 0│  │ 0100 │ 1│ 0100 │ 0011 │00010│00│ 0│0101│      │
│  AL  │  │  │  │  ADD │ S│  R4  │  R3  │  #2 │LSL│  │ R5 │      │
└──────┴──┴──┴──┴──────┴──┴──────┴──────┴─────┴──┴──┴──┴──────┘
                                    └──── Operand2 ────┘

Hex: 0xE0943105
```

ARM Load/Store formát:


```
Nová pole:
- I [25]: 0=immediate offset, 1=register offset
- P [24]: Pre/Post indexing
- U [23]: Add/Subtract offset
- B [22]: Byte/Word (0=word 32bit, 1=byte 8bit)
- W [21]: Write-back (aktualizuj base registr)
- L [20]: Load/Store (1=load, 0=store)
- Rn [19:16]: Base registr (adresa)
- Rd [15:12]: Registr s daty (load do/store z)
- Offset [11:0]: Offset k base registru

Příklad: LDR R1, [R2, #8]
  = Load word z adresy R2+8 do R1
```

Příklad kódování:

```
ADDGT R3, R4, R5    ; R3 = R4 + R5, pouze pokud GT flag

Binárně:
1100 | 00 | 0100 | 0 | 0100 | 0011 | 000000000101
|    |  |    |   |   |      |      |
GT   |  |   ADD  |   R4     R3      R5
     |  └─ data proc
     └─ unconditional
```

Výhody:

- Každá instrukce může být podmíněná (= vysoký výkon, žádný branch overhead)

Nevýhody:

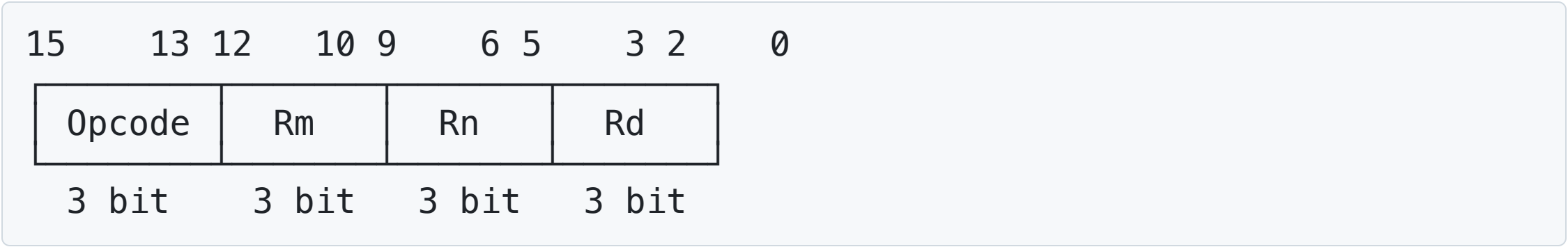
- Velikost kódu vždy 32-bitů (= problém pro embedded s omezenou pamětí)

2.2.2 Thumb režim (16-bitový)

Motivace (mid-1990s):

- Embedded systémy měly 8/16-bit sběrnice
- Flash paměť byla drahá
- Potřeba zmenšit velikost kódu

Charakteristiky:



Omezení:

- Pouze R0-R7 přístupné (z 16 bitů není dost místa)
- Žádné podmíněné vykonávání (až na IT bloky)
- Omezené immediate hodnoty (3-8 bitů)

Příklad:

```
; ARM režim (32-bit každá)
ADD R0, R1, R2          ; 4 byty
MOV R3, #100            ; 4 byty
CMP R4, R5              ; 4 byty
                        ; Celkem: 12 bytů

; Thumb režim (16-bit každá)
ADD R0, R1, R2          ; 2 byty
MOV R3, #100            ; 2 byty
CMP R4, R5              ; 2 byty
                        ; Celkem: 6 bytů (50% úspora!)
```

Výkon vs velikost:

- Kód: ~65% velikosti ARM
- Výkon: ~70% výkonu ARM (více instrukcí potřeba)
- **Sweet spot**: 16-bit sběrnice (žádná penalizace načítání)

Přepínání režimů:

```
; ARM → Thumb
BX Rtarget          ; Branch and eXchange, LSB určuje režim

; Thumb → ARM
BX R14              ; Návrat z Thumb funkce
```

2.2.3 Thumb-2 (16/32-bitový mix)

Inovace (ARMv7-M, 2004):

- Nejlepší z obou světů
- 16-bit instrukce pro jednoduché operace
- 32-bit instrukce pro složitější (ale v Thumb režimu!)

Jak to funguje:

```
16-bit instrukce:
┌──────────────────┐
│ 1110 1xxx xxxx   │ → 16-bit
└──────────────────┘

32-bit instrukce:
┌──────────────────┬──────────────────┐
│ 1111 0xxx xxxx   │ xxxx xxxx xxxx   │ → 32-bit Thumb-2
└──────────────────┴──────────────────┘
```

Dekódování:

- První 16 bitů: pokud začíná 11101 nebo 11110/11111 → 32-bit
- Jinak → 16-bit
- Hardware automaticky pozná

Příklady kódování:

```
; 16-bit Thumb-2
ADDS R0, R1, R2          ; 2 byty, nízké registry
MOV  R2, #5              ; 2 byty, malá immediate

; 32-bit Thumb-2
ADD.W R8, R9, R10        ; 4 byty, vysoké registry
MOV.W R0, #0x12345       ; 4 byty, velká immediate
LDR   R0, [R1, R2, LSL #2]  ; 4 byty, komplexní addressing
```

IT bloky (If-Then):

Thumb-2 nemá podmínění na každé instrukci, místo toho:

```
CMP   R0, R1
ITTEE GT              ; If-Then-Then-Else-Else pokud GT
ADDGT R2, R3, R4      ; Vykonej pokud GT (Then)
MOVGT R5, #1          ; Vykonej pokud GT (Then)
ADDLE R2, R6, R7      ; Vykonej pokud LE (Else)
MOVLE R5, #0          ; Vykonej pokud LE (Else)
```

Výhody Thumb-2:

- Hustota kódu: ~74% ARM (lepší než Thumb)
- Výkon: ~98% ARM (téměř stejný!)
- Eliminace přepínání režimů
- **De facto standard** pro moderní ARM

2.2.4 Srovnání velikosti kódu

Benchmark (reálný embedded kód):

```
Program size:
ARM (32-bit):       100%  (baseline)
Thumb (16-bit):     65%   (menší, ale pomalejší)
Thumb-2 (mix):      74%   (sweet spot)
```

Memory bandwidth:

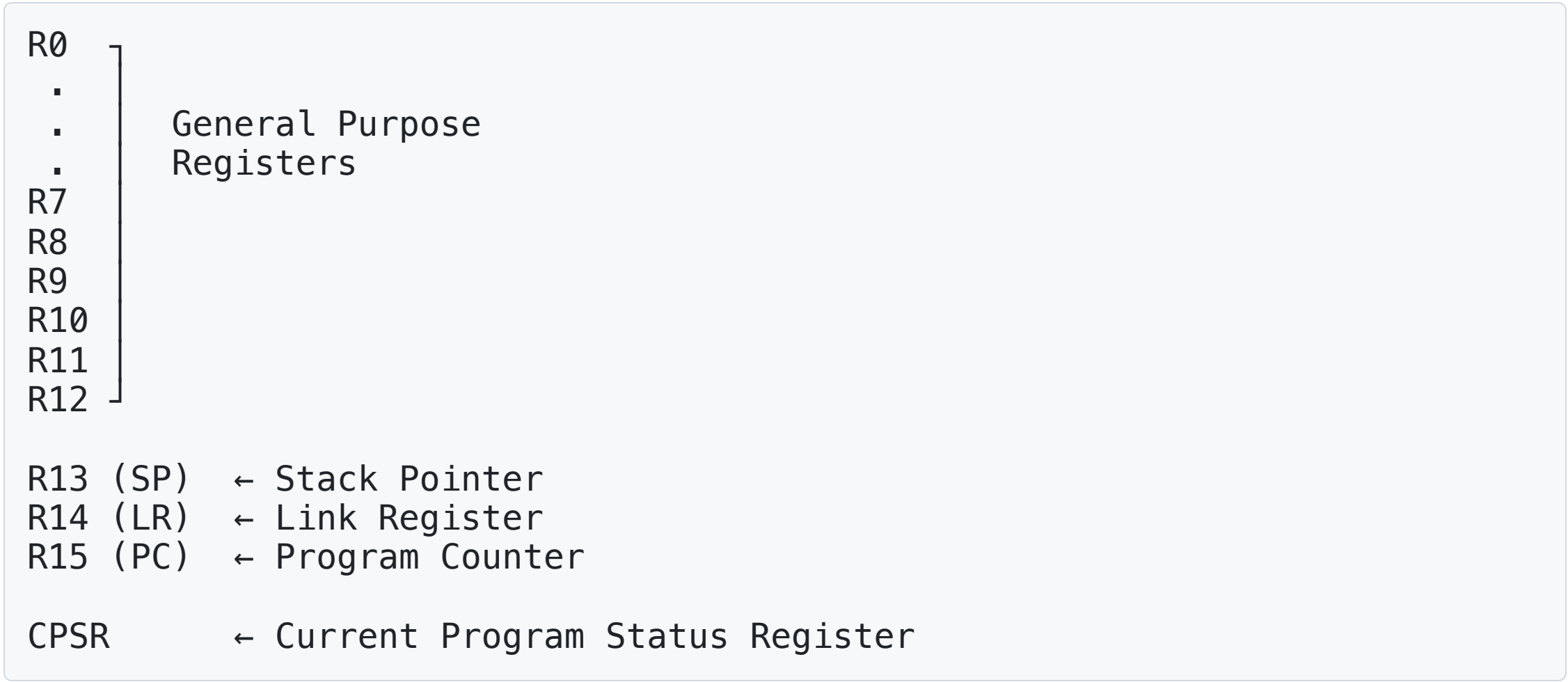
```
ARM:     32-bit fetch každý cyklus
Thumb:   16-bit fetch každý cyklus (menší bandwidth!)
Thumb-2: průměrně 18-20 bit/cyklus
```

2. Architektura ARM

2.3 Organizace registrů

2.3.1 Základní register file (ARMv7)

User/System režim:



Special registers:

- **SP (R13)**: Ukazatel na vrchol zásobníku
  - Každý režim má vlastní SP (FIQ, IRQ, SVC...)
- **LR (R14)**: Link Register

```
BL function    ; PC → LR, skoč na function
...
BX LR          ; Návrat: LR → PC
```

- **PC (R15)**: Program Counter
  - Čtení: PC + 8 (kvůli pipeline)
  - Zápis: skok (branch)

2.3.2 Register banking (režimy procesoru)

ARM má 7 režimů procesoru:

```
User (USR):    Běžné aplikace
System (SYS):  Privilegovaný user mode
FIQ:           Fast Interrupt
IRQ:           Normal Interrupt
Supervisor:    OS kernel, SWI
Abort:         Memory faults
Undefined:     Neplatné instrukce
```

Banky registrů

Registry, které mají více fyzických kopií, ale stejné logické jméno. Aktivní kopie závisí na režimu procesoru.

	User	FIQ	IRQ	SVC	ABT	UND
R0–R7	same	same	same	same	same	same
R8–R12	same	R8_fiq R9_fiq R10_fiq R11_fiq R12_fiq	same	same	same	same
R13(SP)	R13	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und
R14(LR)	R14	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15(PC)	same	same	same	same	same	same
CPSR	same	same	same	same	same	same
SPSR	–	SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

Výhoda FIQ:

- R8-R14 jsou banked → není potřeba ukládat na stack
- Rychlejší vstup do přerušení (Fast Interrupt!)

Příklad IRQ handleru:

```
IRQ_Handler:
    PUSH {R0-R3, R12, LR}   ; Ulož context
    ; ... zpracování přerušení ...
    POP  {R0-R3, R12, PC}   ; Obnov a návrat
    ; SPSR → CPSR automaticky
```

2.3.3 CPSR - Current Program Status Register

Struktura CPSR (32-bit):

- N Výsledek je záporný (Negative)
- Z Výsledek je nula (Zero)
- C Carry/Borrow při aritmetice (Carry)
- V Signed overflow (oVerflow)
- IT [15:10, 26:25] If-Then state (Thumb-2)
- GE [19:16] Greater-Equal flags (SIMD)
- E Endianness (0=little, 1=big)
- A Asynchronous abort disable
- I IRQ disable
- F FIQ disable
- Mode [4:0] Procesor režim

Nastavování příznaků:

```
ADDS R0, R1, R2      ; S suffix: nastaví flags
ADD  R0, R1, R2      ; Bez S: flags nezmění

; Příklad:
MOV R0, #0
SUBS R1, R0, #1      ; R1 = -1
                     ; N=1 (negative)
                     ; Z=0 (not zero)
                     ; C=0 (borrow)
```

2. Architektura ARM

2.4 Podmíněné vykonávání

2.4.1 Princip

Každá ARM instrukce má 4-bitové podmínkové pole:

```
Kód  Sufix  Meaning                Flags
0000  EQ    Equal                  Z=1
0001  NE    Not Equal              Z=0
0010  CS/HS Carry Set              C=1
0011  CC/LO Carry Clear            C=0
0100  MI    Minus/Negative         N=1
0101  PL    Plus/Positive          N=0
0110  VS    Overflow Set           V=1
0111  VC    Overflow Clear         V=0
1000  HI    Higher (unsigned)      C=1 && Z=0
1001  LS    Lower/Same             C=0 || Z=1
1010  GE    Greater/Equal          N=V
1011  LT    Less Than              N!=V
1100  GT    Greater Than           Z=0 && N=V
1101  LE    Less/Equal             Z=1 || N!=V
1110  AL    Always (default)       -
1111  NV    Never (deprecated)     -
```

2.4.2 Eliminace větvení

Klasický RISC (např. MIPS):

```
    CMP R0, R1
    BEQ skip         ; Branch pokud equal
    ADD R2, R3, R4   ; Vykonej pokud not equal
skip:
    ; continue
```

Problém: Branch = pipeline flush, 3-5 cyklů penalizace

ARM s podmíněním:

```
    CMP R0, R1
    ADDNE R2, R3, R4   ; Vykonej pouze pokud NE
    ; continue
```

Výhoda: Žádný branch, žádná penalizace!

2.4.3 Praktické příklady

Absolute value:

```
; Standard způsob (s větvením):
    CMP  R0, #0
    BGE  positive
    NEG  R0, R0
positive:

; ARM podmíněné vykonávání:
    CMP  R0, #0
    NEGLT R0, R0      ; Neguj pouze pokud < 0
```

Max funkce:

```
; R0 = max(R0, R1)
    CMP  R0, R1
    MOVLT R0, R1       ; R0 = R1 pokud R0 < R1
```

Conditional set:

```
    CMP  R0, #10
    MOVGT R1, #1       ; R1 = 1 pokud R0 > 10
    MOVLE R1, #0       ; R1 = 0 pokud R0 <= 10
```

2.4.4 Výhody a nevýhody

Výhody:

- Eliminace krátkých branchů, vyšší výkon, kompaktnější kód
- Deterministický čas vykonání (real-time!)

Nevýhody:

- Komplikovanější dekódování
- Moderní branch prediction je velmi dobrá (>95%)
- V praxi: používá se jen pro krátké sekvence (1-3 instrukce)

Statistika:

- ~10-15% instrukcí v ARM kódu je podmíněných, snížení branchů: ~5-8%
- Celkový přínos výkonu: ~2-5%

2. Architektura ARM

2.5 Praktické aspekty ARM architektury

2.5.1 Calling convention (AAPCS)

ARM Architecture Procedure Call Standard:

```
R0–R3:   Argument/result registers (caller-saved)
R4–R11:  Callee-saved (musí se zachovat)
R12(IP): Intra-Procedure-call scratch
R13(SP): Stack Pointer
R14(LR): Link Register
R15(PC): Program Counter
```

Příklad volání funkce:

```
func_call:
    ; Parametry v R0-R3
    MOV R0, #5          ; arg1
    MOV R1, #10         ; arg2
    BL  my_function     ; Call
    ; Výsledek v R0

my_function:
    PUSH {R4-R5, LR}  ; Ulož callee-saved
    ADD R4, R0, R1    ; Use R4 (callee-saved)
    MOV R0, R4        ; Return value
    POP {R4-R5, PC}   ; Obnov a návrat
```

2.5.2 Pipeline charakteristiky

ARM7 (3-stage):

- CPI: 1.9 typicky (jednoduché)
- Branch penalty: 2 cykly
- Frekvence: 10-100 MHz

ARM9 (5-stage):

- CPI: 1.1-1.3
- Branch penalty: 3 cykly
- Frekvence: 200-450 MHz

Cortex-M3 (3-stage):

- CPI: 1.0-1.3
- Branch: 2-3 cykly
- Obsahuje: branch prediction, prefetch buffer

Cortex-A (8-15 stage):

- CPI: 0.5-1.5 (out-of-order, superskalár)
- Branch: predikce >95% úspěšnost
- Frekvence: 1-3 GHz

2.5.3 Energetická efektivita

Příklad (Cortex-M4 @ 168 MHz):

```
Active mode:    50–80 mW
Sleep mode:     ~10 mW
Deep sleep:     ~100 µW
Standby:        ~5 µW
```

Optimalizační techniky:

- **Clock gating**: Zastavit hodiny nepoužívaných bloků
- **Voltage scaling**: Snížit napětí při nižších frekvencích
- **Sleep modes**: WFI (Wait For Interrupt), WFE (Wait For Event)

3. Architektura RISC-V

3.1 Filozofie a vznik RISC-V

3.1.1 Motivace (2010, UC Berkeley)

Proč vznikl RISC-V? - Existující ISA jsou **proprietární** (ARM, x86, MIPS)

- Licenční poplatky: $1-10 za čip, právní rizika: patentové spory
- Omezení modifikací, nemožnost optimalizovat pro specifické použití
- Vendor lock-in, vysoké vstupní náklady

Cíle RISC-V:

1. **Čistý design**: žádná zpětná kompatibilita
2. **Modulární**: Základní ISA + volitelná rozšíření
3. **Stabilní**: Základní ISA se nikdy nezmění
4. **Škálovatelný**: Od MCU po superpočítače
5. **Open-source**: BSD licence, žádné patenty

3.1.2 Principy architektury

RISC-V = "RISC done right":

- Učení se z 40 let RISC designu
- Odstranění chyb historických RISC (MIPS, SPARC)
- Moderní přístup k ISA designu

Klíčová rozhodnutí:

- **x0 = zero**: Hardwired nulový registr (zjednodušení)
- **Žádné delay sloty**: Na rozdíl od MIPS
- **Žádné podmíněné vykonávání**: Na rozdíl od ARM (komplikuje OoO)
- **Explicitní paměťový model**: Fence instrukce pro ordering
- **Komprimované instrukce**: Volitelné 16-bit (RVC)

3. Architektura RISC-V

3.2 Modulární design - Rozšíření

3.2.1 Základní koncept

Base ISA (povinná):

- **RV32I**: 32-bit integer
- **RV64I**: 64-bit integer
- **RV128I**: 128-bit (experimentální)

Standard rozšíření (volitelná):

```
RV32IMAFDC = RV32I + M + A + F + D + C
           = RV32G (General purpose)
```

3.2.2 Rozšíření M - Multiply/Divide

8 nových instrukcí:

```
MUL    rd, rs1, rs2    # rd = rs1 × rs2 (lower 32 bits)
MULH   rd, rs1, rs2    # rd = (rs1 × rs2) >> 32 (signed)
MULHU  rd, rs1, rs2    # rd = (rs1 × rs2) >> 32 (unsigned)
MULHSU rd, rs1, rs2    # Mixed signed/unsigned

DIV    rd, rs1, rs2    # rd = rs1 / rs2 (signed)
DIVU   rd, rs1, rs2    # Unsigned division
REM    rd, rs1, rs2    # rd = rs1 % rs2 (remainder)
REMU   rd, rs1, rs2    # Unsigned remainder
```

Proč volitelné?

- Mikrořadiče často nemají HW multiplikátor
- Ušetří ~10% plochy čipu
- Software emulace možná (ale 10-100× pomalejší)

Latence:

- MUL: typicky 1-3 cykly (pipelined multiplier)
- DIV: 8-32 cyklů (iterativní algoritmus)

3.2.3 Rozšíření A - Atomic operations

Motivace: Multi-core synchronizace

Load-Reserved / Store-Conditional:

```
LR.W  rd, (rs1)         # Load-Reserved
SC.W  rd, rs2, (rs1)    # Store-Conditional
                        # rd = 0 (success) / 1 (failure)
```

Použití (implementace atomického incrementu):

```
atomic_inc:
    LR.W   t0, (a0)      # Load hodnotu
    ADDI   t0, t0, 1     # Inkrementuj
    SC.W   t1, t0, (a0)  # Zkus uložit
    BNEZ   t1, atomic_inc # Pokud selhalo, opakuj
```

AMO instrukce (Atomic Memory Operations):

```
AMOADD.W  rd, rs2, (rs1)   # Atomic add
AMOSWAP.W rd, rs2, (rs1)   # Atomic swap
AMOAND.W  rd, rs2, (rs1)   # Atomic AND
AMOOR.W   rd, rs2, (rs1)   # Atomic OR
AMOXOR.W  rd, rs2, (rs1)   # Atomic XOR
AMOMIN.W  rd, rs2, (rs1)   # Atomic min
AMOMAX.W  rd, rs2, (rs1)   # Atomic max
```

Memory ordering:

```
AMOADD.W.AQ  rd, rs2, (rs1)  # Acquire semantics
AMOADD.W.RL  rd, rs2, (rs1)  # Release semantics
AMOADD.W.AQRL rd, rs2, (rs1) # Both
```

3.2.4 Rozšíření F a D - Floating Point

F - Single precision (32-bit):

- 32 floating-point registrů (f0-f31)
- IEEE 754-2008 standard
- Instrukce: FADD.S, FSUB.S, FMUL.S, FDIV.S, FSQRT.S

D - Double precision (64-bit):

- Stejné registry f0-f31 (ale 64-bit široké)
- FADD.D, FSUB.D, FMUL.D, FDIV.D

Příklad:

```
FLW   f0, 0(a0)         # Load float z paměti
FLW   f1, 4(a0)
FADD.S f2, f0, f1       # f2 = f0 + f1 (single precision)
FSW   f2, 8(a0)         # Store výsledek
```

Proč oddělené FP registry?

- Paralelismus: Integer a FP instrukce současně
- Větší register file bez tlaku na integer registry
- Specializované FPU jednotky

3.2.5 Rozšíření C - Compressed (16-bit)

Motivace:

- Úspora paměti (jako Thumb u ARM)
- Lepší využití I-cache
- Snížení bandwidth při fetchingu

Princip:

```
Normal:     32-bit instructions
Compressed: 16-bit encoding for common operations
```

Rozpoznání:

```
Bits [1:0]:
  00, 01, 10 → 16-bit compressed instruction
  11         → 32-bit normal instruction
```

Příklady mapování:

```
# 32-bit RV32I          → 16-bit RVC
ADDI x8, x8, 4          → C.ADDI x8, 4       (2 bytes)
LW   x9, 0(x8)          → C.LW   x9, 0(x8)   (2 bytes)
ADD  x10, x9, x11       → C.ADD  x10, x11    (2 bytes)
```

Omezení RVC:

- Pouze x8-x15 pro mnohé instrukce (populární registry)
- Malé immediate hodnoty (5-6 bitů)
- Omezené offsety pro load/store

Výsledky:

- Typická úspora: 25-30% velikosti kódu
- Výkon: téměř identický (dekódování transparentní)

3.2.6 Další rozšíření

B - Bit manipulation:

- Rotace, count leading zeros, parity
- Akcelerace kryptografie a kompresi

V - Vector:

- Proměnná délka vektorů (podobné ARM SVE)
- SIMD operace pro AI/ML

P - Packed SIMD:

- Fixní délka vektorů (128/256-bit)

J - JIT compilation:

- Dynamické linky, JIT support

3. Architektura RISC-V

3.3 RV32I - Base Integer Instruction Set

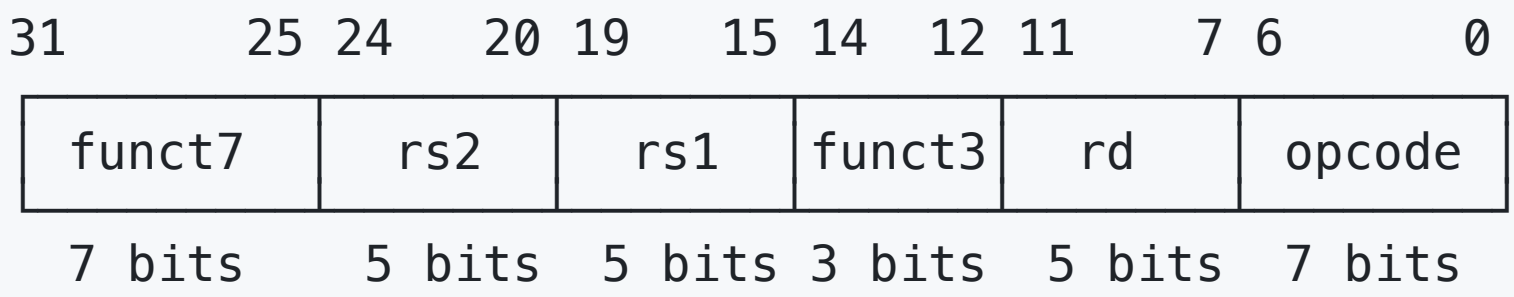
3.3.1 Formáty instrukcí

RISC-V terminologie:

- **opcode**: Základní operační kód (7 bitů) - určuje typ instrukce
- **funct3**: Function 3-bit - upřesnění operace (sub-opcode)
- **funct7**: Function 7-bit - další upřesnění (hlavně u R-type)
- **rd**: Destination register - cílový registr (5 bitů = 32 registrů)
- **rs1**: Source register 1 - první zdrojový registr
- **rs2**: Source register 2 - druhý zdrojový registr
- **imm**: Immediate - konstanta zakódovaná v instrukci

RISC-V má 6 základních formátů:

R-type (Register-Register):



Pole:
- opcode [6:0]: Základní typ (0110011 = OP = ALU reg-reg)
- rd [11:7]: Cílový registr (x0-x31)
- funct3 [14:12]: Funkce (000=ADD/SUB, 111=AND, ...)
- rs1 [19:15]: První zdrojový registr
- rs2 [24:20]: Druhý zdrojový registr
- funct7 [31:25]: Další specifikace (0000000=ADD, 0100000=SUB)

Použití: ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU

Příklad: ADD x3, x1, x2 → x3 = x1 + x2

Dekódování R-type v hardware:

```
Všechna pole jsou na pevných pozicích:
  rs1_addr = instruction[19:15]  // Paralelní extrakce
  rs2_addr = instruction[24:20]
  rd_addr  = instruction[11:7]

Okamžitě můžeme:
  1. Číst rs1 a rs2 z register file
  2. Připravit rd pro zápis
  3. Dekódovat ALU operaci (funct3 + funct7)

Vše v 1 cyklu!
```

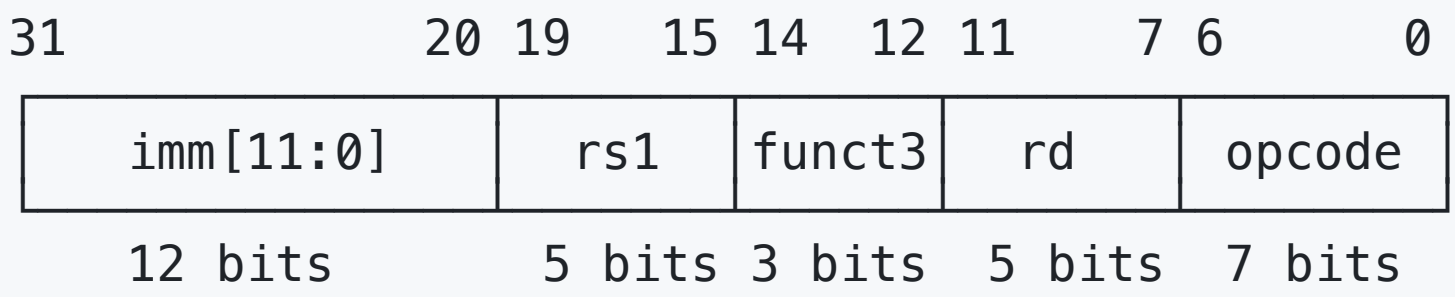
Příklad kompletního R-type kódování:

```
ADD x5, x6, x7

Binární rozpis:
31      25 24   20 19   15 14  12 11    7 6       0
```

0000000	00111	00110	000	00101	0110011
funct7	x7	x6	ADD	x5	OP

```
Hex: 0x007302B3

Dekódování:
- opcode = 0110011 (OP - registr-registr ALU)
- rd = 00101 (x5)
- funct3 = 000 (ADD/SUB)
- rs1 = 00110 (x6)
- rs2 = 00111 (x7)
- funct7 = 0000000 (ADD, ne SUB)

Výsledek: x5 = x6 + x7
```

I-type (Immediate):




```
Pole:
- opcode [6:0]:  OP-IMM (0010011) nebo LOAD (0000011)
- rd [11:7]:     Cílový registr
- funct3 [14:12]: Operace (000=ADDI, 100=XORI, 010=SLTI...)
- rs1 [19:15]:   Zdrojový registr
- imm [31:20]:   12-bit immediate (sign-extended na 32/64 bit)

Použití: ADDI, XORI, ORI, ANDI, SLTI, SLTIU
         LW, LH, LB, LBU, LHU (loads)
         JALR (jump register)

Příklad: ADDI x3, x1, 10 → x3 = x1 + 10
```

Sign-extension immediate:

```
12-bit immediate: imm[11:0]
→ Sign-extended na 32-bit:
  imm_32 = { {20{imm[11]}}, imm[11:0] }

Příklad:
  imm = 0x00A (10 v decimálním)
    = 000000001010 (12 bitů)
  → 0x0000000A (32 bitů)

  imm = 0xFFE (-2 v decimálním, two's complement)
    = 111111111110 (12 bitů)
  → 0xFFFFFFFE (32 bitů, stále -2)
```

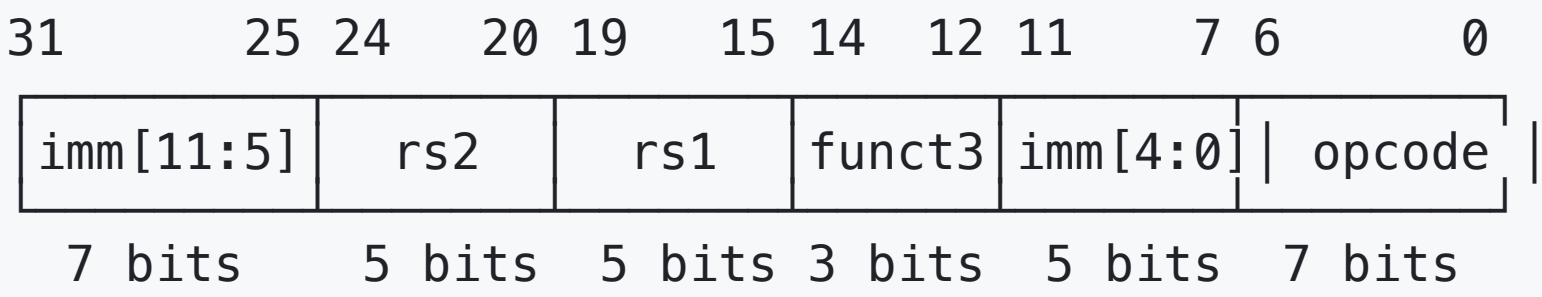
Proč immediate na bitech [31:20]?

```
Design choice pro jednoduché dekódování:
- Bit 31 (sign bit) na nejvyšší pozici
- Hardware může okamžitě začít sign-extension
- Paralelně s dekódováním rs1, rd

Alternative (immediate na jiné pozici) by vyžadovala:
  1. Extrahovat immediate z jiných bitů
  2. Rekonstruovat sign bit
  3. Pak teprve sign-extend
→ Pomalejší!
```

Příklad I-type kódování:

```
ADDI x10, x11, -100

Binární:
31            20 19   15 14  12 11    7 6       0
┌────────────────┬───────┬──────┬───────┬─────────┐
│ 111110011100   │ 01011 │ 000  │ 01010 │ 0010011 │
│  -100 (12bit)  │  x11  │ ADDI │  x10  │ OP-IMM  │
└────────────────┴───────┴──────┴───────┴─────────┘

Hex: 0xF9C58513

Immediate výpočet:
  -100 v decimálním = -0x64
  12-bit two's complement:
    100 = 0000 0110 0100
    NOT = 1111 1001 1011
    +1  = 1111 1001 1100 (0xF9C)
```

S-type (Store):




```
Pole:
- opcode [6:0]:   STORE (0100011)
- imm[4:0] [11:7]:  Dolních 5 bitů offsetu
- funct3 [14:12]:   Velikost (000=SB, 001=SH, 010=SW)
- rs1 [19:15]:      Base registr (adresa)
- rs2 [24:20]:      Source registr (data k uložení)
- imm[11:5] [31:25]: Horních 7 bitů offsetu

Použití: SW, SH, SB (stores)

Příklad: SW x2, 8(x1) → mem[x1+8] = x2
```

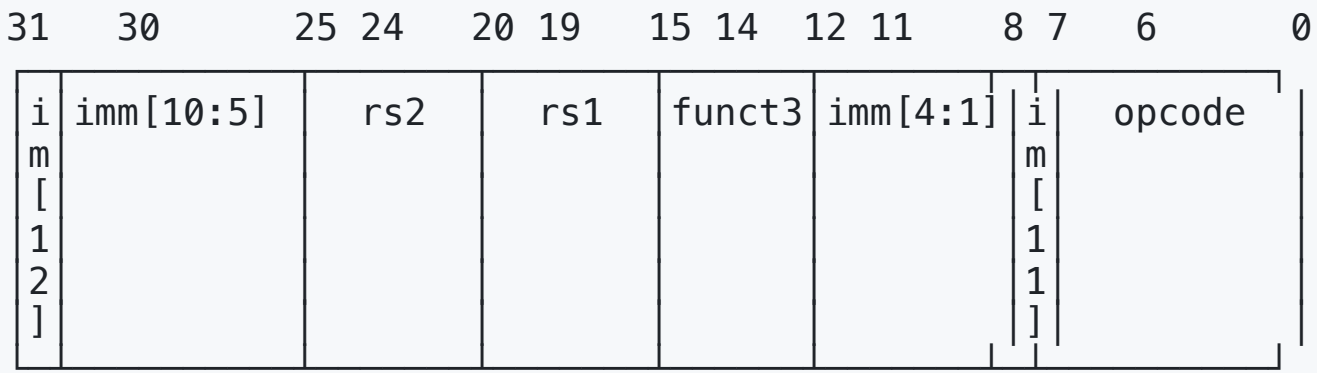
Proč rozdělený immediate?

```
Důvod: Zachovat rs1, rs2 na stejných pozicích jako R-type!

R-type:  rs1[19:15], rs2[24:20], rd[11:7]
S-type:  rs1[19:15], rs2[24:20], imm split na [11:7] a [31:25]

→ Dekodér může použít stejný hardware pro:
  - Extrakci rs1, rs2 (nezávisle na typu instrukce)
  - Register file read ports připojené na stejná místa

Rekonstrukce immediate:
  imm[11:0] = {instruction[31:25], instruction[11:7]}

Jednoduchý multiplexer v hardware!
```

Příklad S-type kódování:

```
SW x5, 12(x10)  # Store word x5 do mem[x10+12]

Binární:
31      25 24   20 19   15 14  12 11    7 6       0
┌─────────┬───────┬───────┬──────┬────────┬─────────┐
│ 0000000 │ 00101 │ 01010 │ 010  │ 01100  │ 0100011 │
│imm[11:5]│  x5   │  x10  │  SW  │imm[4:0]│ STORE   │
│   (0)   │ (rs2) │ (rs1) │      │  (12)  │         │
└─────────┴───────┴───────┴──────┴────────┴─────────┘

Immediate 12 = 0000 0000 1100
  imm[11:5] = 0000000 (bity [31:25])
  imm[4:0]  = 01100   (bity [11:7])

Hex: 0x00552623
```

B-type (Branch):




```
Pole:
- opcode [6:0]:    BRANCH (1100011)
- imm[11] [7]:     Bit 11 offsetu
- imm[4:1] [11:8]: Bity 4–1 offsetu
- funct3 [14:12]:  Podmínka (000=BEQ, 001=BNE, ...)
- rs1 [19:15]:     První porovnávaný registr
- rs2 [24:20]:     Druhý porovnávaný registr
- imm[10:5] [30:25]: Bity 10–5 offsetu
- imm[12] [31]:    Bit 12 (sign bit) offsetu

Offset je 2-byte aligned → imm[0] je vždy 0 (nekóduje se)

Použití: BEQ, BNE, BLT, BGE, BLTU, BGEU

Příklad: BEQ x1, x2, offset → if(x1==x2) PC += offset
```

Proč tak složité immediate?

```
Rekonstrukce:
  imm[12:1] = {inst[31], inst[7], inst[30:25], inst[11:8]}
  imm[0] = 0 (vždy, instrukce jsou 2-byte aligned)

Výsledek: 13-bit offset, sign-extended na 32/64 bit
  Rozsah: -4096 až +4094 bytů

Proč takhle?
1. Zachovat rs1, rs2 na stejných pozicích
2. Sign bit [31] na správném místě pro sign-extension
3. Offset bity strategicky rozmístěny pro minimum HW

Výsledná adresa:
  target_address = PC + sign_extend(offset)
```

Příklad B-type kódování:

```
BEQ x5, x6, 100  # Skoč na PC+100 pokud x5==x6

Offset 100 = 0x64 = 0000 0110 0100

B-type immediate (bit 0 není):
  [12:1] = 0 0000 0110 010

Rozmístění v instrukci:
  imm[12]   = 0          → bit[31]
  imm[10:5] = 000001     → bits[30:25]
  imm[4:1]  = 1001       → bits[11:8]
  imm[11]   = 0          → bit[7]

Binární:
31   30      25 24   20 19   15 14  12 11    8 7   6      0
┌─┬──────────┬──────┬──────┬──────┬──────┬─┬──────────┐
│0│ 000001   │ 00110│ 00101│ 000  │ 1001 │0│ 1100011  │
└─┴──────────┴──────┴──────┴──────┴──────┴─┴──────────┘

Hex: 0x02628463
```

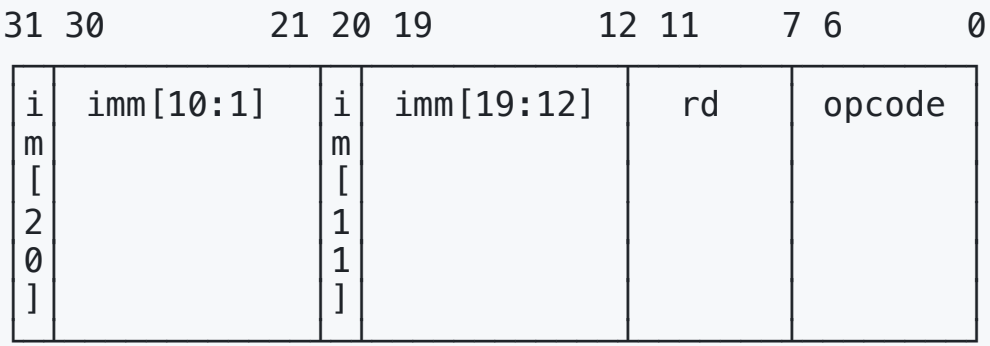
U-type (Upper Immediate):


```
Pole:
- opcode [6:0]:  LUI (0110111) nebo AUIPC (0010111)
- rd [11:7]:     Cílový registr
- imm[31:12]:    Horních 20 bitů immediate

LUI: Load Upper Immediate
  rd = imm << 12 (dolních 12 bitů = 0)

AUIPC: Add Upper Immediate to PC
  rd = PC + (imm << 12)

Použití: Načtení velkých konstant (kombinace s I-type)

Příklad: LUI x3, 0x12345 → x3 = 0x12345000
```

Jak načíst 32-bit konstantu?

```
# Chceme: x5 = 0x12345678

LUI   x5, 0x12345    # x5 = 0x12345000
ADDI  x5, x5, 0x678  # x5 = 0x12345000 + 0x678 = 0x12345678

# Dvě instrukce, ale můžeme načíst libovolnou 32-bit hodnotu!

Pozor na záporné offsety:
# Chceme: x5 = 0x12345FFF

LUI   x5, 0x12346    # x5 = 0x12346000 (zaokrouhlit nahoru!)
ADDI  x5, x5, -1     # x5 = 0x12346000 - 1 = 0x12345FFF
```

J-type (Jump):




```
Pole:
- opcode [6:0]:     JAL (1101111)
- rd [11:7]:        Cílový registr (return address)
- imm[19:12] [19:12]: Bity 19-12 offsetu
- imm[11] [20]:     Bit 11 offsetu
- imm[10:1] [30:21]: Bity 10-1 offsetu
- imm[20] [31]:     Bit 20 (sign bit)

JAL: Jump And Link
  rd = PC + 4 (uložit návratovou adresu)
  PC = PC + sign_extend(offset)

Offset je 2-byte aligned → imm[0]=0

Použití: Volání funkcí, dlouhé skoky

Příklad: JAL x1, offset → x1=PC+4; PC+=offset
```

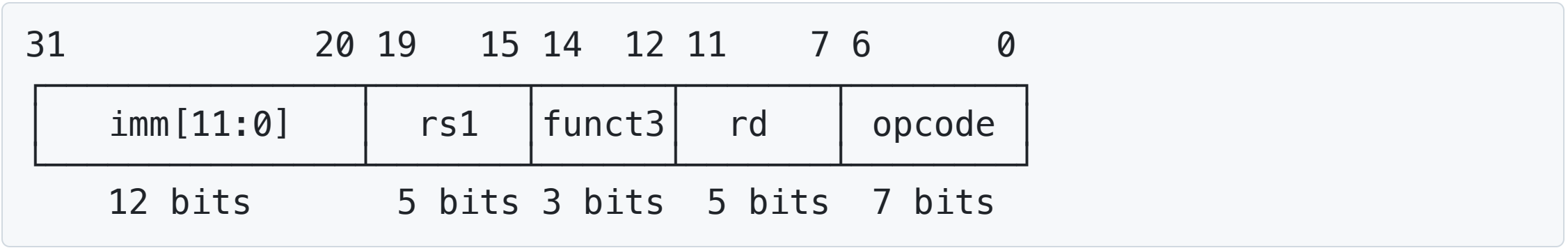
Proč JAL má tak velký rozsah?

```
J-type offset: 21 bitů (imm[20:0], bit 0 vždy 0)
  Rozsah: ±1 MiB (−1048576 až +1048574 bytů)

B-type offset: 13 bitů
  Rozsah: ±4 KiB (−4096 až +4094 bytů)

Důvod:
- Funkce bývají daleko od sebe
- Branching jen v rámci funkce (krátké vzdálenosti)
- JAL používán pro mezifunkční skoky

Kombinace JAL + JALR umožňuje skok kamkoliv:
  AUIPC x1, %pcrel_hi(target)
  JALR  x1, %pcrel_lo(target)(x1)
```

Srovnání: proč RISC-V formáty jsou lepší než x86?

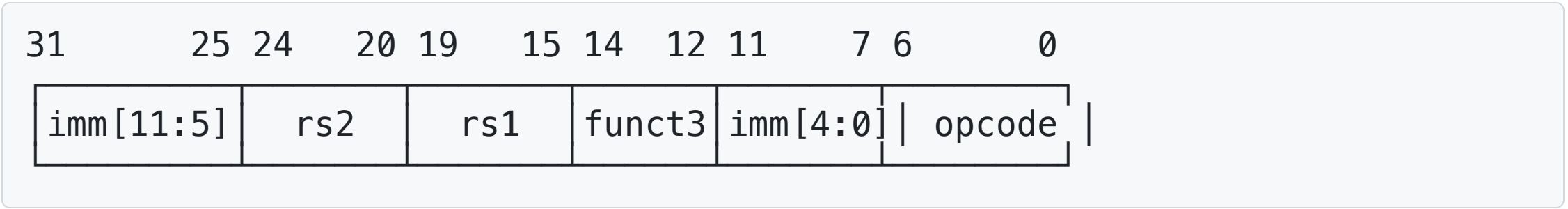
```
RISC-V:
  ✓ Pevná pozice rs1, rs2, rd ve všech formátech
  ✓ Paralelní dekódování možné
  ✓ Jednoduchý hardware
  ✓ Immediate strategicky rozmístěné
  ✓ Sign bit vždy na bit[31]

x86:
  ✗ Proměnná délka (1–15 bytů)
  ✗ Sekvenční dekódování nutné
  ✗ Složitý front-end (µop cache, predecode)
  ✗ Registry na různých pozicích

Výsledek:
  RISC-V dekodér: ~10K gates
  x86 dekodér: ~100K gates (10× složitější!)
```

I-type (Immediate):



Příklad: `ADDI x3, x1, 10` → x3 = x1 + 10

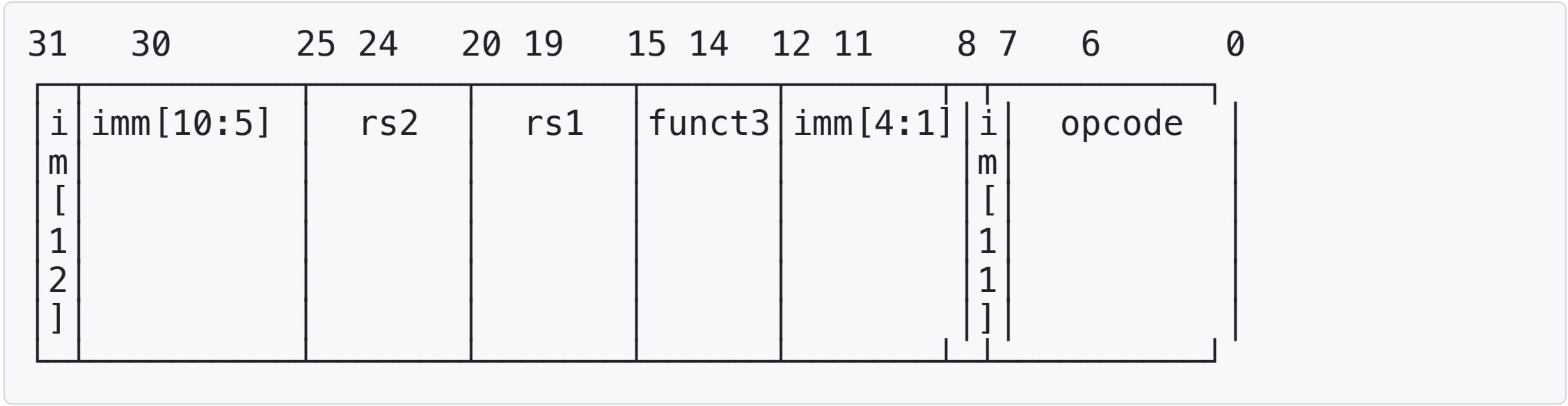
S-type (Store):



Immediate rozdělený (design pro dekódování!)

Příklad: `SW x2, 8(x1)` → mem[x1+8] = x2

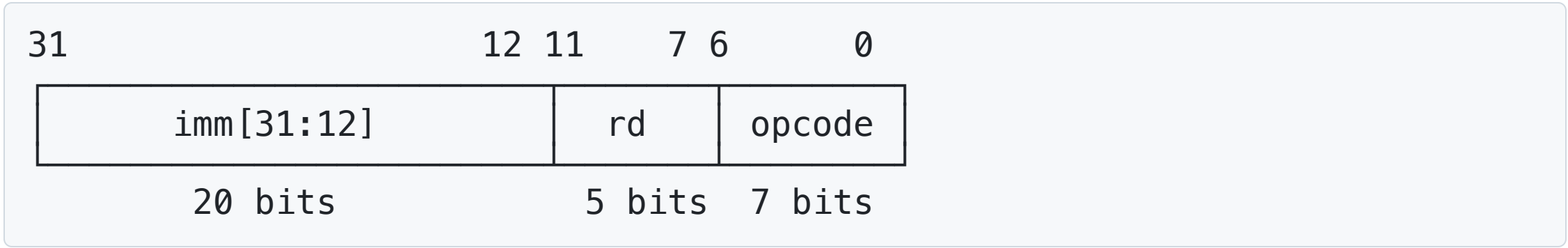
B-type (Branch):



Branch offset (2-byte aligned)

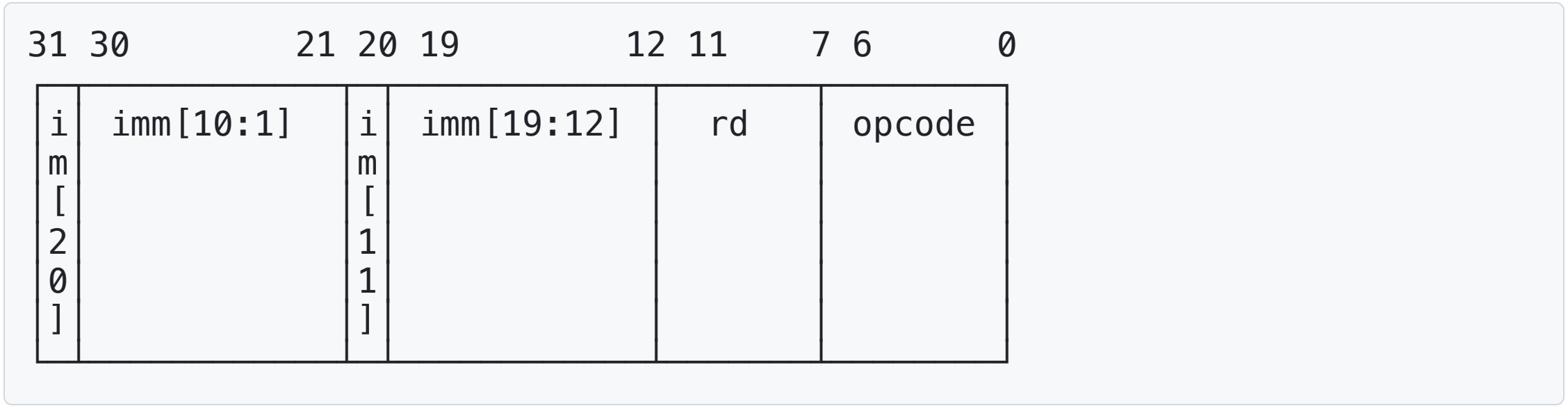
Příklad: `BEQ x1, x2, offset` → if(x1==x2) PC += offset

U-type (Upper Immediate):



Příklad: `LUI x3, 0x12345` → x3 = 0x12345000

J-type (Jump):



Příklad: `JAL x1, offset` → x1 = PC+4, PC += offset

Proč složité immediate kódování?

- Maximální re-use hardware: rs1, rs2, rd na stejných pozicích
- Dekodér může paralelně extrahovat všechna pole
- Immediate znaménkové rozšíření vždy z bitu 31

3.3.2 Registry

32 celočíselných registrů:

```
x0   (zero)  Hardwired zero (čtení vždy vrací 0, zápis ignorován)
x1   (ra)    Return address
x2   (sp)    Stack pointer
x3   (gp)    Global pointer
x4   (tp)    Thread pointer
x5–7 (t0–2)  Temporaries (caller-saved)
x8   (s0/fp) Saved register / Frame pointer
x9   (s1)    Saved register
x10–11(a0–1) Function args/return values
x12–17(a2–7) Function arguments
x18–27(s2–11)Saved registers (callee-saved)
x28–31(t3–6) Temporaries
```

x0 = zero - geniální myšlenka?

```
# NOP (no operation)
ADDI x0, x0, 0        # x0 = x0 + 0 (nic nedělá)

# Unconditional branch
BEQ x0, x0, label     # vždy skočí (x0 vždy == x0)

# Clear register
ADD x5, x0, x0        # x5 = 0

# Test equal to zero
BEQ x5, x0, zero_case # if (x5 == 0)

# Copy register
ADDI x5, x6, 0        # x5 = x6 + 0 = x6
```

Eliminuje potřebu speciálních instrukcí!

3.3.3 RV32I Instrukce (47 total)

Integer aritmetika:

```
ADD   rd, rs1, rs2    # rd = rs1 + rs2
SUB   rd, rs1, rs2    # rd = rs1 - rs2
ADDI  rd, rs1, imm    # rd = rs1 + imm (sign-extended)

# Porovnání
SLT   rd, rs1, rs2    # rd = (rs1 < rs2) ? 1 : 0 (signed)
SLTU  rd, rs1, rs2    # Unsigned compare
SLTI  rd, rs1, imm
SLTIU rd, rs1, imm
```

Logické operace:

```
AND   rd, rs1, rs2
OR    rd, rs1, rs2
XOR   rd, rs1, rs2
ANDI  rd, rs1, imm
ORI   rd, rs1, imm
XORI  rd, rs1, imm
```

Shifts:

```
SLL   rd, rs1, rs2    # Shift left logical (rs2 = shift amount)
SRL   rd, rs1, rs2    # Shift right logical (zero fill)
SRA   rd, rs1, rs2    # Shift right arithmetic (sign extend)
SLLI  rd, rs1, shamt  # Immediate shifts (shamt = 5 bits)
SRLI  rd, rs1, shamt
SRAI  rd, rs1, shamt
```

Load/Store:

```
LW    rd, offset(rs1)   # Load word (32-bit)
LH    rd, offset(rs1)   # Load halfword (16-bit, sign-extend)
LHU   rd, offset(rs1)   # Load halfword unsigned
LB    rd, offset(rs1)   # Load byte (8-bit, sign-extend)
LBU   rd, offset(rs1)   # Load byte unsigned

SW    rs2, offset(rs1)  # Store word
SH    rs2, offset(rs1)  # Store halfword
SB    rs2, offset(rs1)  # Store byte
```

Branches (conditional):

```
BEQ   rs1, rs2, offset  # Branch if equal
BNE   rs1, rs2, offset  # Branch if not equal
BLT   rs1, rs2, offset  # Branch if less than (signed)
BGE   rs1, rs2, offset  # Branch if greater/equal
BLTU  rs1, rs2, offset  # Unsigned comparisons
BGEU  rs1, rs2, offset
```

Jumps (unconditional):

```
JAL   rd, offset         # Jump and link
                         # rd = PC + 4, PC += offset
JALR  rd, rs1, offset    # Jump and link register
                         # rd = PC + 4, PC = rs1 + offset
```

Upper immediate:

```
LUI   rd, imm            # Load upper immediate
                         # rd = imm << 12
AUIPC rd, imm            # Add upper immediate to PC
                         # rd = PC + (imm << 12)
```

System:

```
ECALL                    # Environment call (syscall/trap)
EBREAK                   # Breakpoint (debugger)
FENCE                    # Memory ordering
FENCE.I                  # Instruction fence (I-cache sync)
```

3.3.4 Kódování příkladů

Příklad 1: ADDI x5, x6, 10

```
Instrukce: ADDI x5, x6, 10
Format: I-type

31             20 19   15 14  12 11    7 6       0
┌─────────────────┬───────┬──────┬───────┬─────────┐
│  000000001010   │ 00110 │ 000  │ 00101 │ 0010011 │
│    imm=10       │ rs1=6 │func3 │ rd=5  │ opcode  │
└─────────────────┴───────┴──────┴───────┴─────────┘

Hex: 0x00A30293
```

Příklad 2: ADD x7, x8, x9

```
Instrukce: ADD x7, x8, x9
Format: R-type

31       25 24   20 19   15 14  12 11    7 6       0
```

0000000	01001	01000	000	00111	0110011
funct7	rs2=9	rs1=8	funct3	rd=7	opcode

```
Hex: 0x009403B3
```

Příklad 3: LW x10, 12(x11)

```
Instrukce: LW x10, 12(x11)
Format: I-type

31             20 19   15 14  12 11    7 6       0
┌─────────────────┬───────┬─────┬────────┬─────────┐
│  000000001100   │ 01011 │ 010 │ 01010  │ 0000011 │
│   offset=12     │ base=11 width │ rd=10  │ LOAD    │
└─────────────────┴───────┴─────┴────────┴─────────┘

Hex: 0x00C5A503
```

3. Architektura RISC-V

3.4 RV64I - 64-bit rozšíření

Registry mají šířku 64b (W - word, D - doubleword)

Nové instrukce (12 přidaných):

```
# 64-bit arithmetic
ADDW  rd, rs1, rs2    # 32-bit add, sign-extend result to 64
SUBW  rd, rs1, rs2
ADDIW rd, rs1, imm

# 64-bit shifts on 32-bit values
SLLW  rd, rs1, rs2    # Shift lower 32 bits
SRLW  rd, rs1, rs2
SRAW  rd, rs1, rs2
SLLIW rd, rs1, shamt
SRLIW rd, rs1, shamt
SRAIW rd, rs1, shamt

# Load/Store
LD    rd, offset(rs1)  # Load doubleword (64-bit)
SD    rs2, offset(rs1) # Store doubleword
LWU   rd, offset(rs1)  # Load word unsigned (zero-extend to 64)
```

3. Architektura RISC-V

3.5 Privilege Levels - Úrovně oprávnění

3.5.1 Tři režimy

M-mode (Machine):

- Nejvyšší privilegia, vždy přítomen
- Plný přístup k hardware
- Boot code, firmware, M-mode runtime (OpenSBI)
- **Použití**: Embedded systémy často běží pouze v M-mode

S-mode (Supervisor):

- OS kernel (Linux, FreeBSD)
- Řízení virtuální paměti (page tables)
- Může zachytit U-mode excepty
- **Volitelný**: Embedded systémy ho nemusí mít

U-mode (User):

- Aplikace bez privilegií
- Nemůže měnit CSR registry
- Nemůže manipulovat s pamětí OS
- **Volitelný**: Embedded systémy ho nemusí mít

Přechody mezi režimy:

```
U-mode ←→ S-mode ←→ M-mode
(trap)    (trap)
(return)  (return)
```

3.5.2 Control and Status Registers (CSR)

M-mode CSRs:

```
mstatus    Machine status register
misa       ISA and extensions (R/W or read-only)
mie        Machine interrupt enable
mtvec      Machine trap vector base address
mscratch   Scratch register for M-mode trap handler
mepc       Machine exception program counter
mcause     Machine trap cause
mtval      Machine trap value (faulting address/instruction)
mip        Machine interrupt pending
```

S-mode CSRs:

```
sstatus    Supervisor status
sie        Supervisor interrupt enable
stvec      Supervisor trap vector
sscratch   Supervisor scratch
sepc       Supervisor exception PC
scause     Supervisor trap cause
stval      Supervisor trap value
sip        Supervisor interrupt pending
satp       Supervisor address translation and protection
```

CSR instrukce:

```
CSRRW  rd, csr, rs1    # rd = CSR; CSR = rs1 (atomic swap)
CSRRS  rd, csr, rs1    # rd = CSR; CSR |= rs1 (set bits)
CSRRC  rd, csr, rs1    # rd = CSR; CSR &= ~rs1 (clear bits)
CSRRWI rd, csr, imm    # Immediate versions
CSRRSI rd, csr, imm
CSRRCI rd, csr, imm
```

Příklad: Disable interrupts

```
CSRRCI x0, mstatus, 0x8   # Clear MIE bit in mstatus
# x0 = destination (discard old value)
```

3.5.3 Trap handling

Trap = Exception nebo Interrupt:

- **Exception**: Synchronous (illegal instruction, page fault)
- **Interrupt**: Asynchronous (timer, external device)

Trap flow:

```
1. HW automaticky:
   - mepc = PC (uložit návratovou adresu)
   - mcause = trap kód
   - mtval = doplňující info
   - Skoč na mtvec (trap handler)

2. Software trap handler:
   - Ulož context (registry)
   - Zpracuj trap (podle mcause)
   - Obnov context
   - MRET (Machine RETurn)
```

mcause kódy:

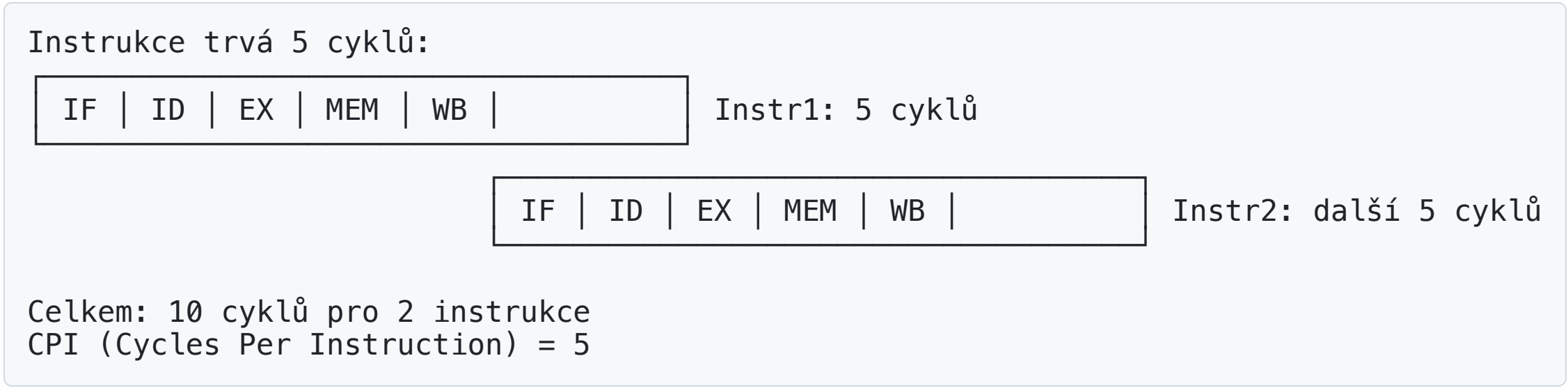
```
Interrupt bit (bit 63/31) = 1 → Interrupt
                          = 0 → Exception

Exception codes:
0:  Instruction address misaligned
1:  Instruction access fault
2:  Illegal instruction
3:  Breakpoint
4:  Load address misaligned
5:  Load access fault
6:  Store address misaligned
7:  Store access fault
8:  Environment call from U-mode
9:  Environment call from S-mode
11: Environment call from M-mode
12: Instruction page fault
13: Load page fault
15: Store page fault
```

4. Implementace zřetězení (Pipelining)

4.1 Motivace: Proč pipeline?

4.1.1 Sekvenční vykonávání (bez pipeline)

Jednoduchý procesor:



Problém: Hardware je většinu času nečinný!

- Když EX počítá, IF/ID/MEM/WB nedělají nic
- Plýtvání prostředky

4.1.2 Pipeline: Paralelní zpracování

S pipeline:

```
Čas:  1    2    3    4    5    6    7    8    9
I1:  |IF |ID |EX |MEM|WB |
I2:       |IF |ID |EX |MEM|WB |
I3:            |IF |ID |EX |MEM|WB |
I4:                 |IF |ID |EX |MEM|WB |

Celkem: 9 cyklů pro 4 instrukce
Po naplnění: 1 instrukce/cyklus → CPI = 1
```

Výhody pipeline:

- Vyšší throughput (propustnost)
- Lepší využití hardware
- Vyšší IPC (Instructions Per Cycle)

Nevýhody:

- Složitější řízení
- Hazardy (konflikty)
- Vyšší latence jednotlivé instrukce

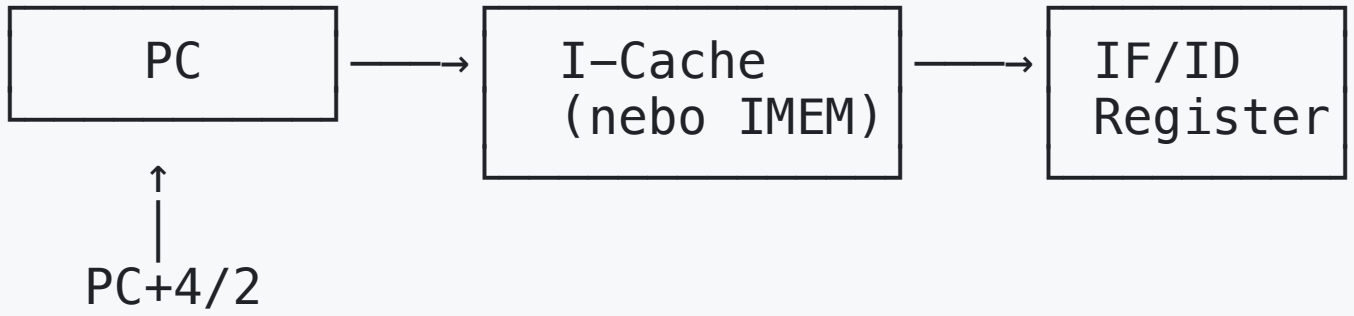
4. Implementace zřetězení (Pipelining)

4.2 Klasické 5stupňové zřetězení

4.2.1 Fáze pipeline - Detailní rozbor

IF (Instruction Fetch) - Načtení instrukce:

```
Operace:
1. Čti PC (Program Counter)
2. Přistup do I-Cache/paměti
3. Načti instrukci
4. PC = PC + 4 (nebo PC + 2 pro compressed)
5. Ulož do IF/ID registru

Hardware:
```



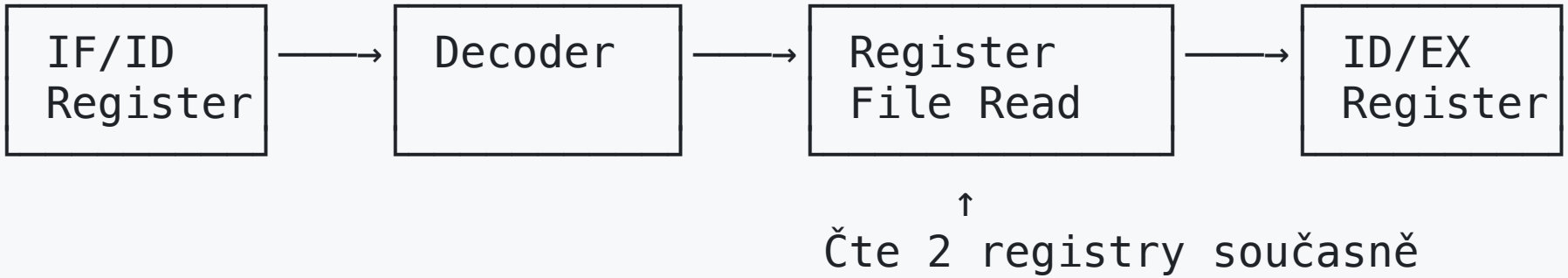

```
Latence: 1 cyklus (s cache hit)
```

Možné problémy:

- I-Cache miss: +10-100 cyklů
- Branch: PC se může změnit

ID (Instruction Decode) - Dekódování:

```
Operace:
1. Dekóduj opcode z instrukce
2. Identifikuj registry (rs1, rs2, rd)
3. Čti registry z register file
4. Dekóduj immediate hodnotu
5. Generuj řídicí signály
6. Ulož do ID/EX registru

Hardware:
```



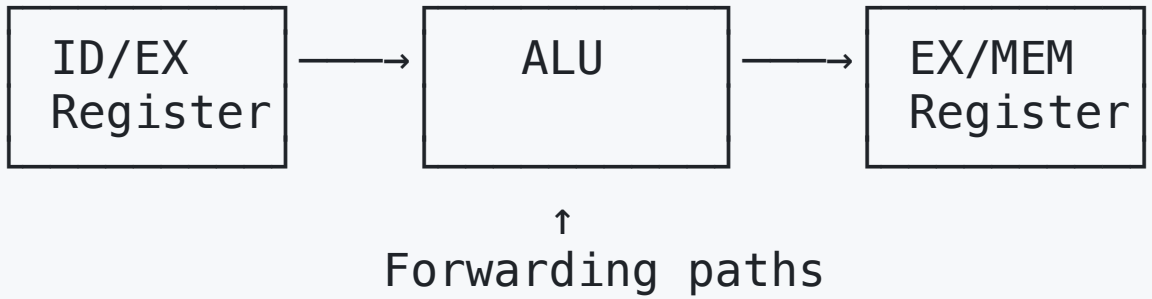

```
Latence: 1 cyklus
```

Dekódování pro různé ISA:

```
RISC-V (pevná délka):
  - rs1 vždy na bitech [19:15]
  - rs2 vždy na bitech [24:20]
  - rd vždy na bitech [11:7]
  → Paralelní dekódování!

x86 (proměnná délka):
  - Musíme sekvenčně číst byty
  - Složitější dekódování
  - Proto x86 používá předekódování
```

EX (Execute) - Vykonání:

```
Operace:
1. Vyber operandy (z registrů nebo immediate)
2. Provedení v ALU:
   - Aritmetika (ADD, SUB, MUL)
   - Logika (AND, OR, XOR)
   - Shift (SLL, SRL, SRA)
   - Porovnání (SLT, SLTU)
3. Pro load/store: výpočet adresy
4. Pro branch: vyhodnocení podmínky + cílová adresa
5. Ulož výsledek do EX/MEM registru

Hardware:
```



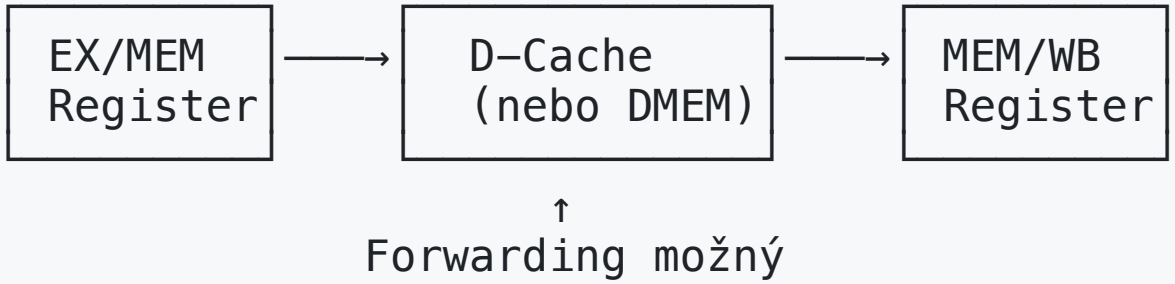

```
Latence: 1 cyklus (pro většinu operací)
```

Komplikace:

- Násobení: může trvat 2-3 cykly (multi-cycle)
- Dělení: 8-32 cyklů (iterativní)
- Floating-point: 3-5 cyklů

MEM (Memory Access) - Přístup do paměti:

```
Operace:
1. Pro LOAD:
   - Použij adresu z EX
   - Přistup do D-Cache
   - Načti data
2. Pro STORE:
   - Použij adresu z EX
   - Zapiš data z registru do D-Cache
3. Pro ostatní:
   - Nic (data jdou přímo do WB)
4. Ulož do MEM/WB registru

Hardware:
```



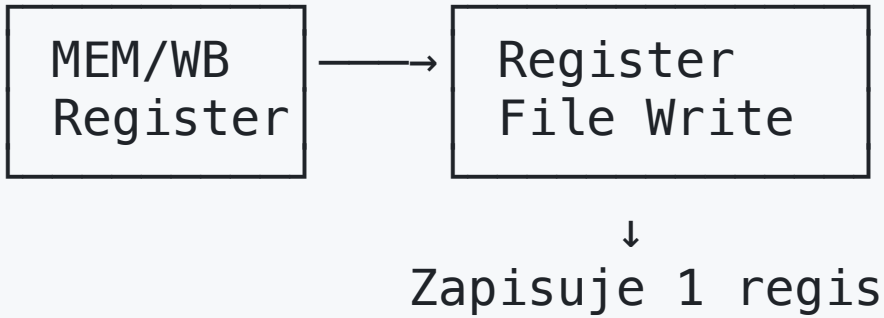

```
Latence: 1 cyklus (s cache hit)
```

Možné problémy:

- D-Cache miss: +10-100 cyklů
- Alignment error: adresa není zarovnaná
- Page fault: data nejsou v RAM (OS swap)

WB (Write Back) - Zápis zpět:

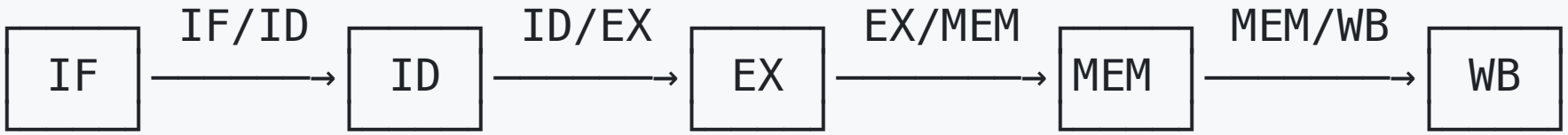
```
Operace:
1. Vezmi výsledek z MEM/WB registru
2. Zapiš do cílového registru (rd)
3. Pro některé instrukce: nic (store, branch)

Hardware:
```



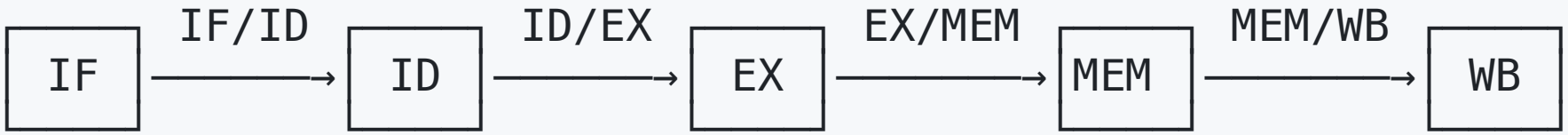

```
Latence: 1 cyklus (polovina cyklu - write v 1. půli)
```

4.2.2 Pipeline registry

Mezi každým stupněm:




```
Obsah IF/ID registru:
  - Instrukce (32-bit)
  - PC (pro výpočet branch target)

Obsah ID/EX registru:
  - Operand1, Operand2 (hodnoty z registrů)
  - Immediate
  - rs1, rs2, rd (čísla registrů)
  - Řídicí signály (ALU op, mem read/write, ...)
  - PC
```




```
Obsah EX/MEM registru:
  - ALU výsledek
  - Data pro store
  - rd
  - Řídicí signály (mem read/write, reg write)

Obsah MEM/WB registru:
  - Data z paměti nebo ALU
  - rd
  - Řídicí signály (reg write)
```

Proč registry mezi stupni?

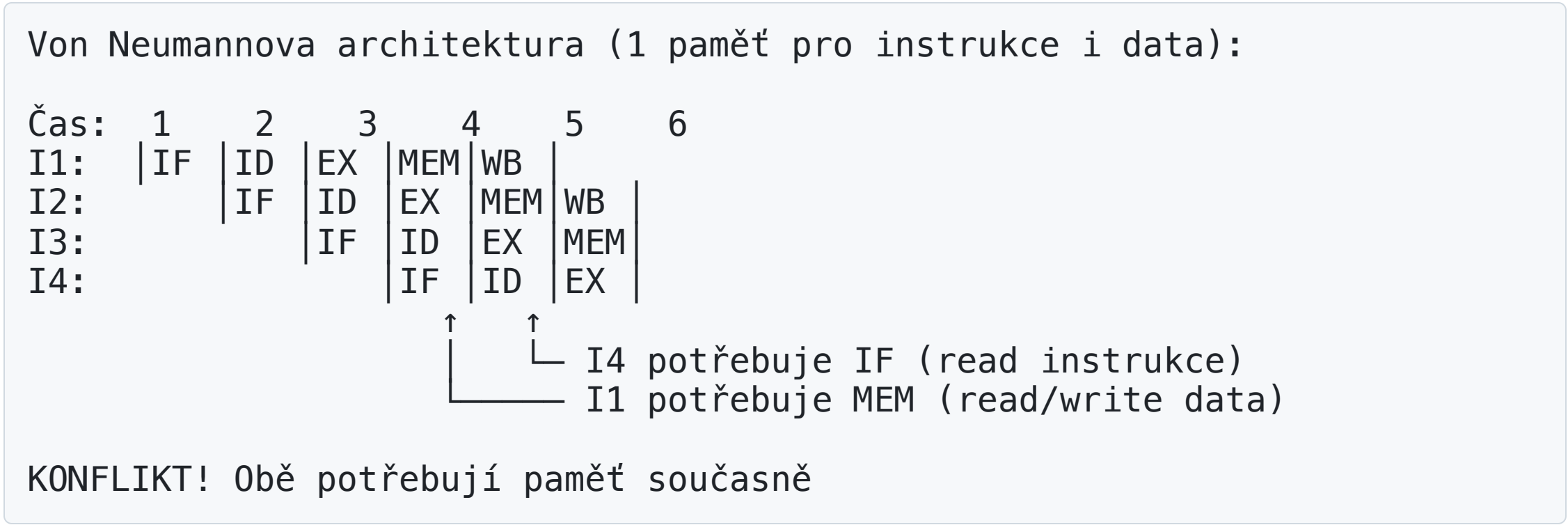
- Synchronizace: všechny fáze trvají 1 cyklus
- Izolace: každá fáze vidí "snapshot" dat
- Umožňují paralelismus

4. Implementace zřetězení (Pipelining)

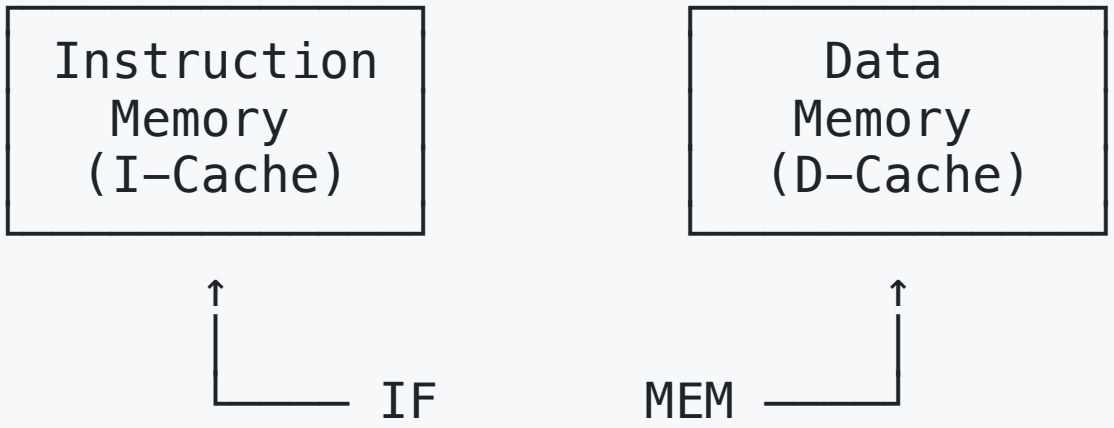
4.3 Hazardy pipeline - Typy a řešení

4.3.1 Strukturální hazardy - konflikt o HW prostředky

Příklad 1: Sdílená paměť



Řešení: Harvardská architektura

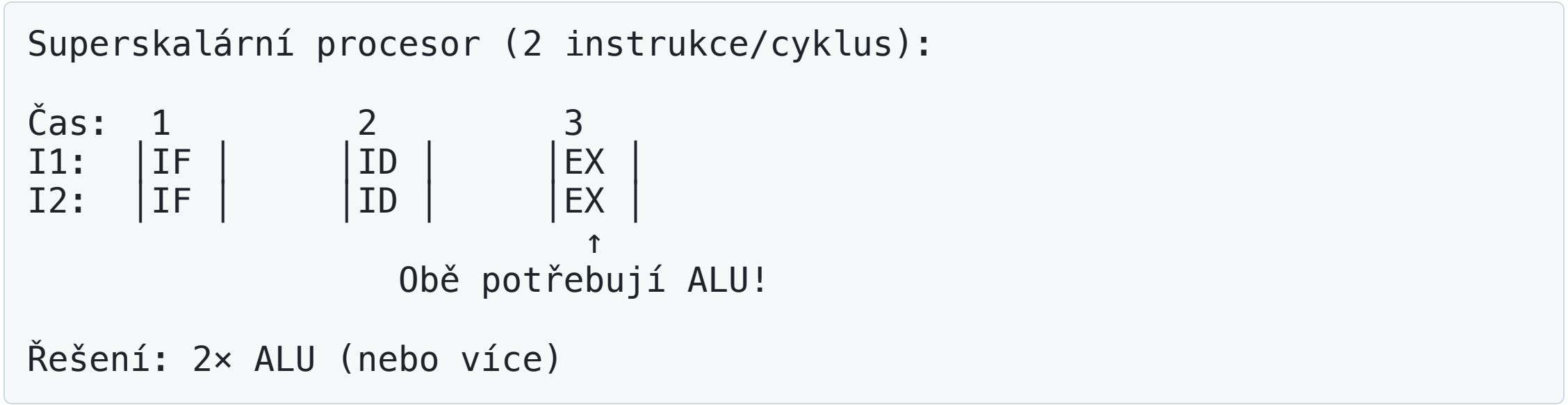


→ Žádný konflikt, obě mohou běžet paralelně

Příklad 2: Register file konflikt

```
Problém: Co když ID stage chce číst registr,
         který WB stage právě zapisuje?

Řešení: Register file s internal forwarding
  - Write v 1. půli cyklu
  - Read v 2. půli cyklu
  - Nebo: bypass write→read v rámci registru
```

Příklad 3: ALU není dost



4.3.2 Datové hazardy - Read After Write (RAW)

Definice: Instrukce potřebuje výsledek předchozí instrukce

Příklad 1: Základní RAW hazard

```
ADD x1, x2, x3    # x1 = x2 + x3
SUB x4, x1, x5    # x4 = x1 - x5  ← potřebuje x1!
```

Časový diagram (bez řešení):

```
Čas:  1    2    3    4    5    6    7
ADD: │IF │ID │EX │MEM│WB │
SUB:      │IF │ID │EX │MEM│WB │
               ↑
               └─ SUB čte x1 v ID (cyklus 3)

ADD zapisuje x1 ve WB (cyklus 5)
→ SUB dostane STAROU hodnotu x1!
→ CHYBA!
```

Shrnutí: Klíčové poznatky o pipeline

1. **Pipeline zvyšuje throughput, ne latenci**
    - Jedna instrukce stále trvá stejně
    - Ale dokončujeme více instrukcí per čas
2. **Hazardy jsou nevyhnutelné**
    - Datové (RAW): forwarding + stalling
    - Řídicí (branch): predikce
    - Strukturální: více hardware
3. **CPI je klíčová metrika**
    - Ideální: CPI = 1.0, reálné: CPI = 1.1-2.0
    - Závisí na: hazardech, predikci, cache