Mikroarchitektura a řízení operací

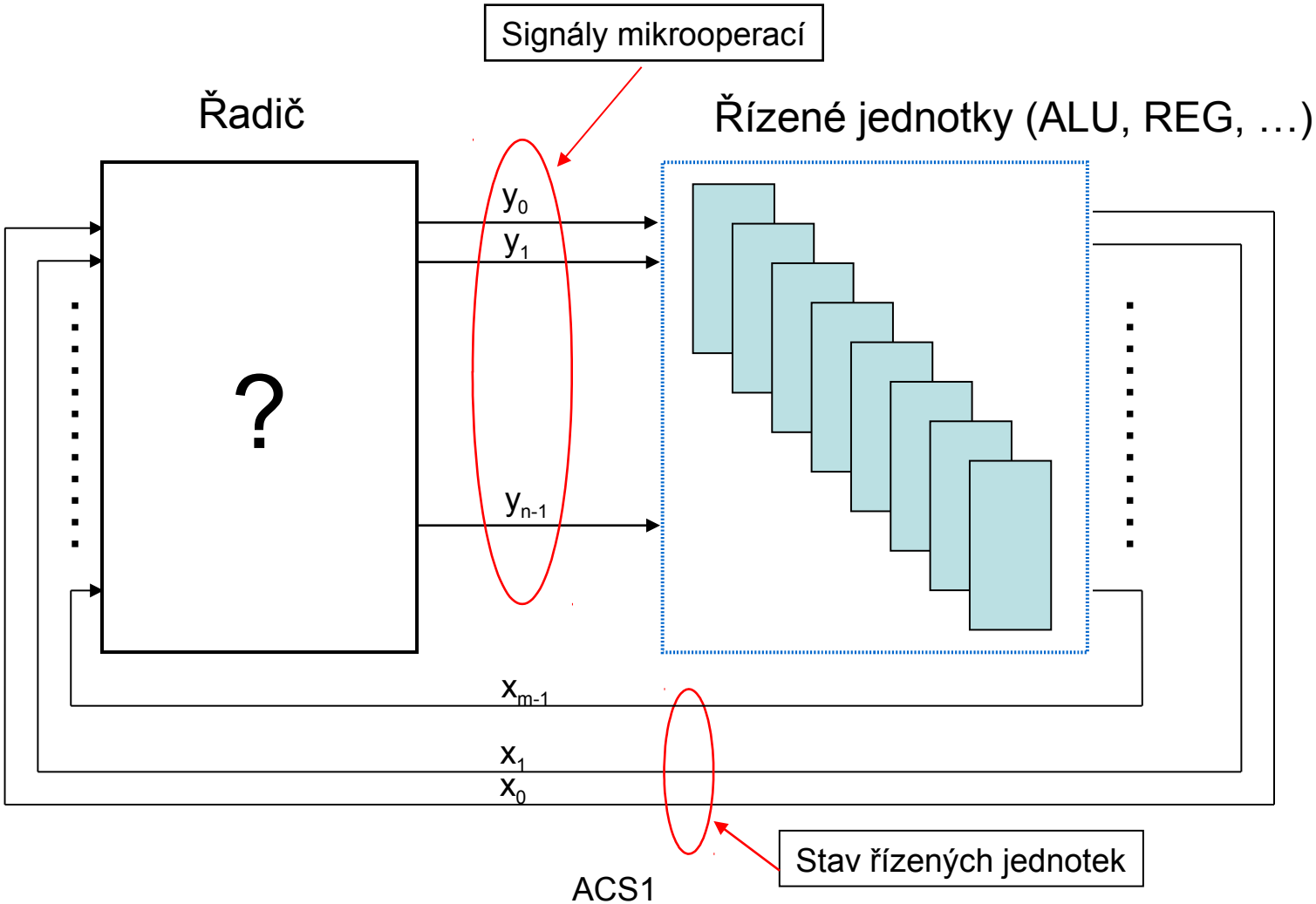
Mikroarchitektura

- Úkolem mikroarchitektury je implementace ISA (Instruction Set Architecture).
- Jedna ISA může mít větší množství implementací, které se navzájem liší.
- Tato vrstva počítačového systému leží nad úrovní digitální logiky.
- „Doba života“ jedné verze mikroarchitektury bývá jen zlomkem trvání ISA.

Provádění instrukcí

- Proces vykonání instrukce není jednorázový děj, instrukce se vykonává ve fázích
- Postupné provádění jednotlivých fází instrukcí zajišťuje řadič procesoru
  - automat
  - patří do třídy sekvenčních logických obvodů

Řídící jednotka (řadič)



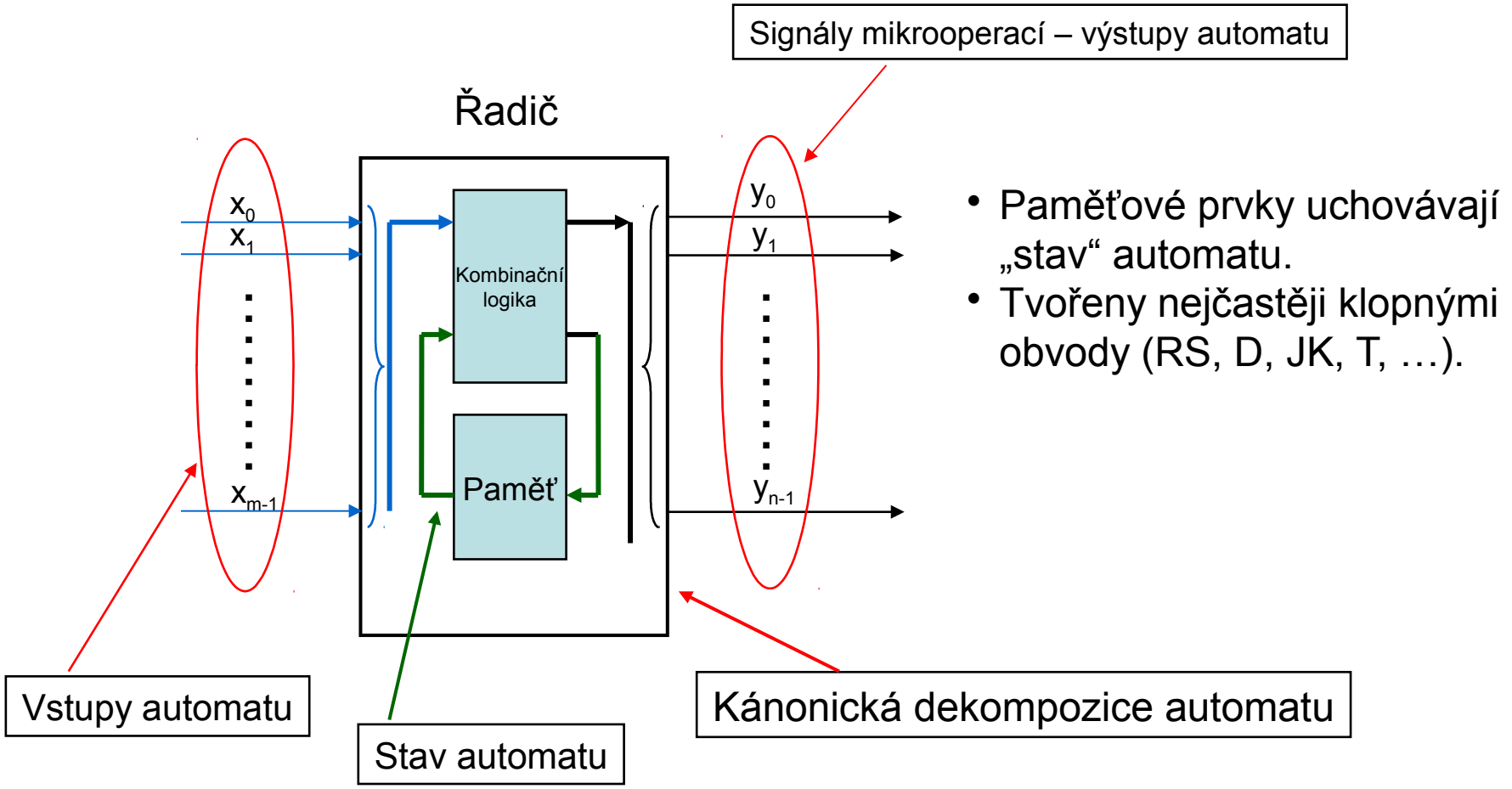
Struktura řídící jednotky

- Řídící jednotka
  - „Pevný automat“ (přímé HW řízení)
    - Mooreův automat
    - Mealyho automat
  - Mikroprogramový automat (Wilkes-Stringer) (jedná se o nadstavbu pevných automatů)

Pozn: V minulosti byly používány i různé exotické varianty řídících jednotek:

- Zpožďovací linky (n*LC), určené pro generaci posloupností řídících signálů
- Zapojení monostabilních FF, kopírujících vývojové algoritmy řídících algoritmů, místo rozhodovacích bloků - signálové výhybky …

Řídící jednotka – pevný automat



- Paměťové prvky uchovávají „stav“ automatu.
- Tvořeny nejčastěji klopnými obvody (RS, D, JK, T, …).

Struktura řídící jednotky: pokračování

- Podle toho, kdy dochází ke změně stavu automatu rozlišujeme:
  - Automaty synchronní
    - Okamžik změny výstupu a stavu určen hranou hodinového signálu, vstupní signály určují charakter změny (spolu s aktuálním stavem)
    - Kromě vstupních signálů automat využívá i hodinový kmitočet, který pochází z oscilátoru.
  - Automaty asynchronní
    - Okamžik změny výstupu a stavu odvozován jen od změn vstupních signálů

Pro asynchronní automaty je typické použití asynchronních klopných obvodů (RS, …)

Struktura řídící jednotky: pokračování

- Mealyho a Mooreův automat se od sebe liší jen způsobem generování výstupních signálů:

Mealy: $s(t+1) = f[x(t), s(t)]$
$y(t) = g[x(t), s(t)]$

Moore: $s(t+1) = f[x(t), s(t)]$
$y(t) = g[s(t)]$

Pozn. 1: Neplést stavové proměnné řídícího automatu a signály o stavu řízených jednotek !!!

Pozn. 2: Synchronní verze Mooreova automatu mění výstupy pouze po synchronizační hraně hodin, Mealyho automat kdykoliv, protože mezi jeho vstupy a výstupy leží jen kombinační síť !

Struktura řídící jednotky: pokračování

- Pevné automaty se vyznačují velmi rychlou reakcí na podněty a jsou vhodné pro implementaci jednodušších a středně složitých algoritmů řízení.
- S rozvojem technologie a hlavně návrhových vývojových prostředků se tyto hranice neustále posouvají.
- Mikroprogramové automaty jsou doménou počítačů třídy CISC.

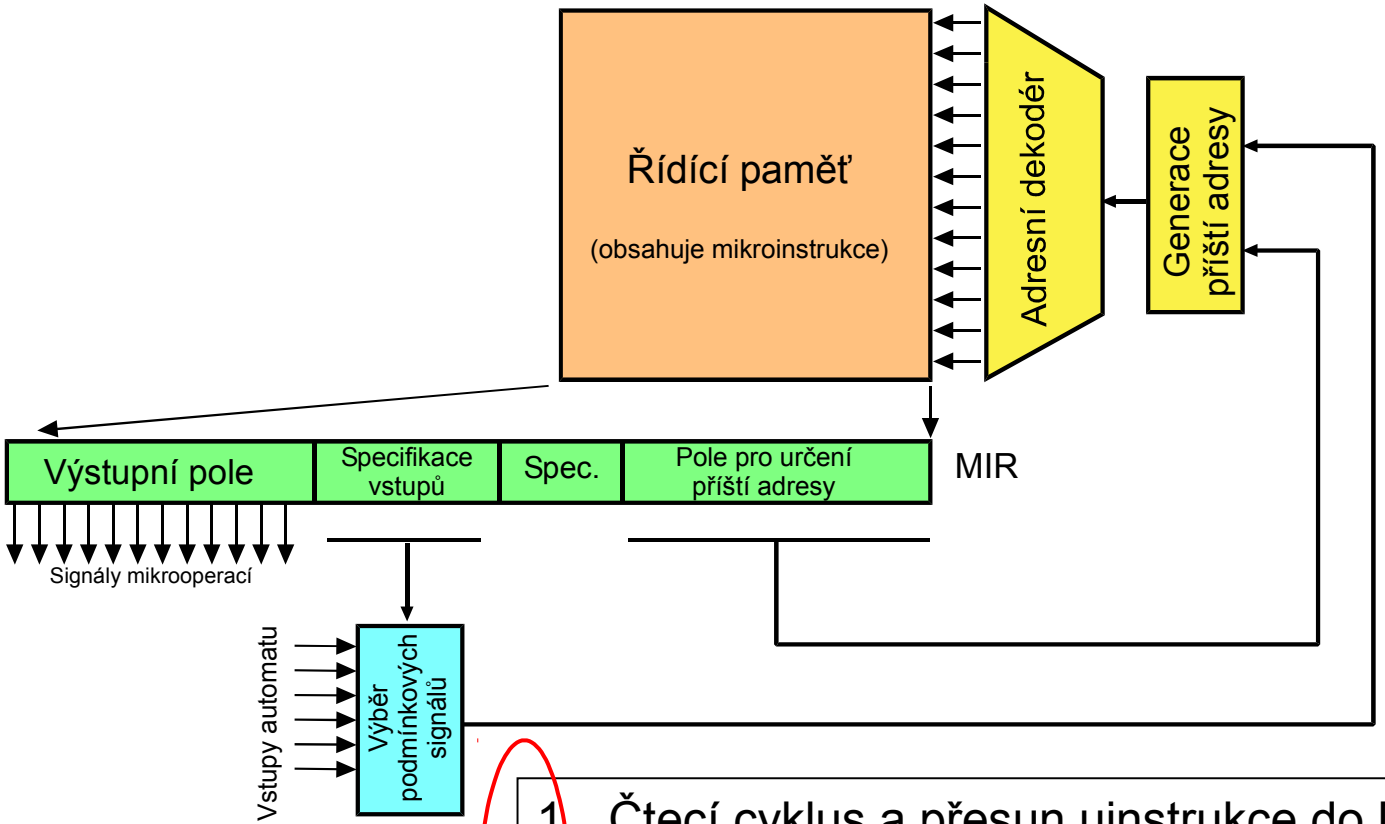
Maurice V. Wilkes

Narozen 26.6.1913, Staffordshire, UK

1967 Turing Award citace:

Profesor Wilkes je znám jako tvůrce a návrhář EDSACu, prvního počítače s interně uloženým programem. EDSAC byl postaven v r. 1949 a používal jako paměť rtuťovou zpožďovací linku. Také je znám (spolu s Wheelerem a Gillem) jako autor svazku “Preparation of Programs for Electronic Digital Computers” z roku 1951, kde bylo uvedeno efektivní využití programových knihoven.

Struktura Wilkesova automatu



Tento cyklus řídí jednoduchý „pevný“ automat

1. Čtecí cyklus a přesun uinstrukce do MIR
2. Dekódování a generace výstupních signálů
3. Výběr podmínkových signálů a modifikace adresy
4. Výběr příští uinstrukce

Struktura řídící jednotky: pokračování

- Přestože princip mikroprogramového řízený byl znám již od 50-tých letech minulého století, jeho nasazení bránil nedostatek vhodných dostatečně rychlých pamětí pro uložení mikroprogramu.
- Nejčastěji se používaly paměti organizace 2D, pracující na různých fyzikálních principech:
  - diodové matice
  - odporové matice (Tesla 200)
  - tranzistorové matice
  - feritové jádrové paměti (! zachování stejné indukčnosti)
  - transformátorové paměti (ICL xxx, Odra 1204)
  - matice s kapacitní vazbou (poslední modely řady IBM360)
  - matice s induktivní vazbou
  - a mnoho dalších ……

Mikroprogramování

- Letmý pohled na počítače s mikroprogramovým řízením. Důvod:
  - Ukázat jak stavět velmi malé procesory s komplexní ISA.
  - Pomoci porozumět jak stroje CISC vznikly.
  - Stále je používáno u řady strojů (x86, PowerPC, …).
  - Poslouží jako „jemný“ úvod do vnitřní struktury stroje.
  - Pomůže pochopit, jak technologie přispěla k přechodu na systémy RISC.

Mapování ISA do mikroarchitektury

- ISA je často navržena pro určitý styl mikroarchitektury, např.:
  - CISC ⇒ mikroprogramové řízení
  - RISC ⇒ hardwarové řízení, pipelinning
  - VLIW ⇒ fixní latence ve strukturách v režimu pipeline
  - JVM ⇒ softwarová interpretace
- Určitá ISA může být implementována v libovolném stylu mikroarchitektury
  - Core 2 Duo: hardwarově řízený stroj CISC (x86) s mikroprogramovou podporou
  - Tato lekce: mikroprogramově řízený stroj RISC (MIPS)
  - Intelský dynamicky plánovaný „out-of-order“ VLIW (IA-64) procesor
  - PicoJava: Hardwarový JVM procesor

Složitost firmware

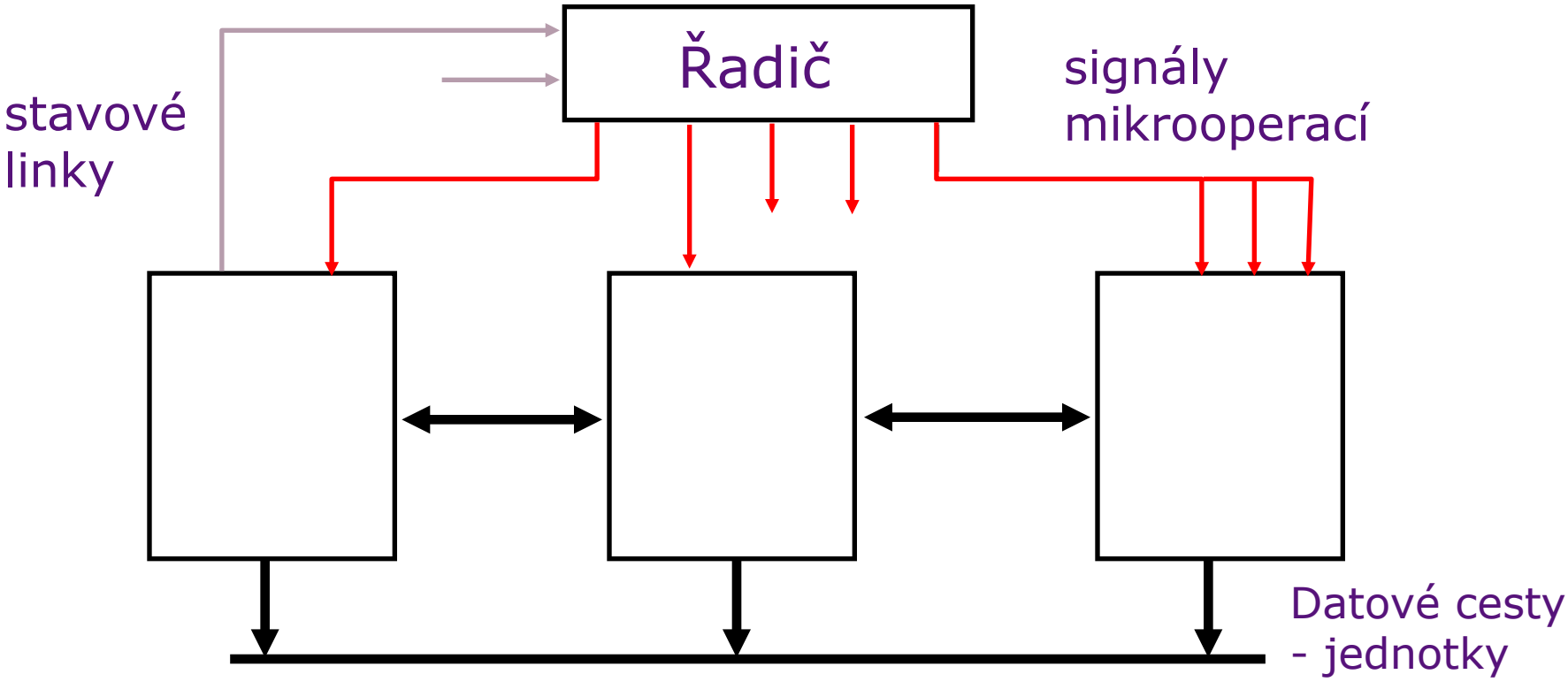
- Přístup při vytváření mikroprogramů se liší od klasického vytváření programů:
  - Návrháři firmware musí respektovat *mikroskopickou* složitost
  - Návrháři firmware se snaží vytvářet *strojově nezávislý* model

  } Protichůdné požadavky

- Specifika mikroarchitektury jsou přímou příčinou složitosti firmware. Sem patří následující problémy:
  - Propojitelnost částí mikroprogramu
  - Paralelizmus
  - Charakteristiky časování
  - Speciální vlastnosti registrů a funkčních jednotek (neregularity)

Snahou je umožnit implementaci mikroprogramové úrovně pomocí programovacích jazyků vyšší úrovně.

Mikroarchitektura: *Implementace určitého typu ISA*

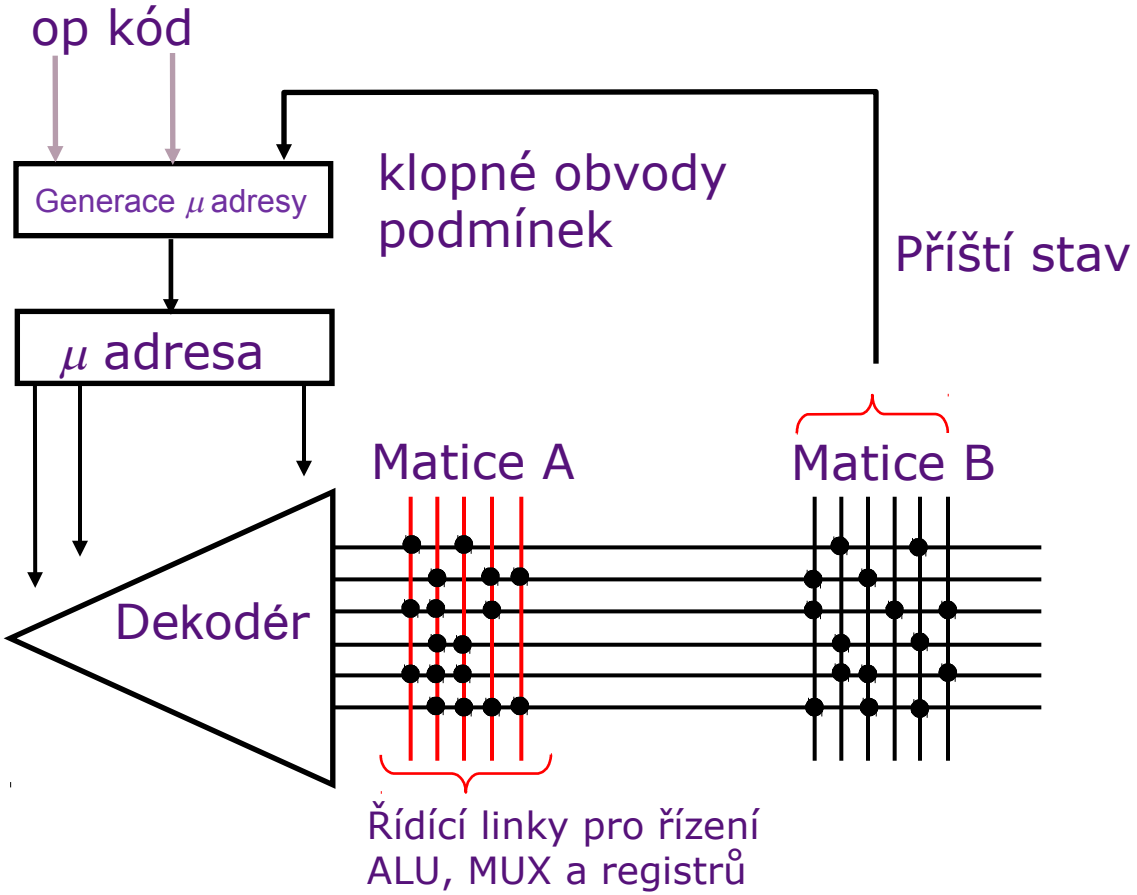


Struktura: Jak jsou komponenty propojeny.
Statické

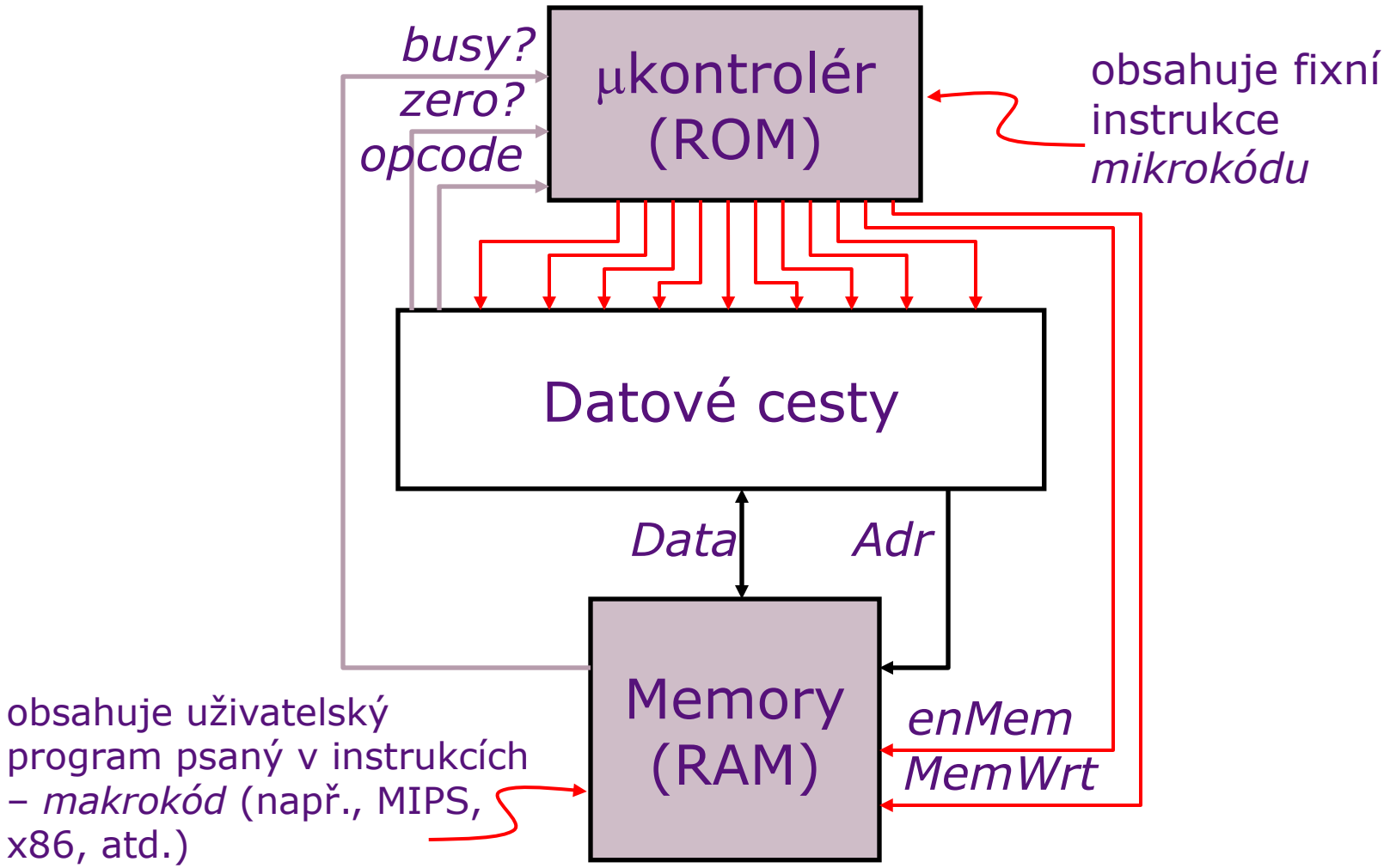
Chování: Jak jsou data mezi komponentami přenášena
Dynamické

Microprogramový řadič: *Maurice Wilkes, 1954*

Stavová tabulka řídící logiky vestavěna do paměť. pole



Mikrokódovaná mikroarchitektura



Pojmy

- **Mikrooperace** – elementární, dále již nedělitelná operace, kterou uvozuje příslušný „signál mikrooperace“ (inkrementace čítače, nulování registru, zápis dat do registru, …).
- **Mikroinstrukce** – mikroinstrukční slovo, získané čtením řídící paměti v mikroinstrukčním cyklu. Udává, které mikrooperace se mají během mikroinstrukčního cyklu provést.
- **Mikroprogram** – posloupnost mikroinstrukcí, které interpretují jednotlivé instrukce instrukčního souboru, nebo jejich částí.
- **Firmware** – „mikroprogramové vybavení“ procesoru.

Pozn.: Řídící paměť bývá velmi často typu ROM, v některých případech RAM. Pak je nutno před zahájením činnosti procesoru řídící paměť naplnit firmwarem.

Výstupy mikroprogramového automatu

- Časování mikroprogramového automatu hraje podstatnou roli ve funkci celého procesoru.
- Z hlediska práce jednotlivých funkčních jednotek procesoru lze rozdělit výstupní signály na dvě velké skupiny:
  - Výskyt signálu pouze v rámci jedné mikroinstrukce.
  - Signály s trváním delším než jeden mikroinstrukční takt (např. nastavení operace ALU, nastavení multiplexerů, řízení „pomalejších“ jednotek).

Mikroinstrukční pole

- Mikroinstrukce obsahuje několik polí, které jsou během jejího zpracování vyhodnoceny a je odvozena příslušná aktivita.
  - **Operační kód mikroinstrukce** – rozlišuje jednotlivé typy mikroinstrukcí.
  - **Výstupní pole** – obsahuje informace o signálech mikrooperací, které mají být během daného mikroinstrukčního cyklu aktivní.
  - **Specifikace vstupů** – určuje specifikace vstupních podmínkových signálů, které slouží k větvení mikroprogramu.
  - **Komparační bit** – slouží pro inverzi podmínky.
  - **Pole příští adresy** – v závislosti na strategii volby příští mikroinstrukce obsahuje jednu nebo více adres, na které bude mikroprogram pokračovat.
  - **Ochranné bity** – například parita.
  - **Spec. pole** – diagnostické účely.

Pozn.: Všechna pole nemusí být vždy v mikroinstrukci obsažena.

Generace adresy příští mikroinstrukce

Strategie (příklady):

- **Využití mPC** – analogie PC na mikroprogramové úrovni. Využívá sekvenční charakter mikroprogramů. Podle vybrané podmínky se postoupí na další mikroinstrukci a nebo se provede skok v mikroprogramu.
- **Výběr adresy** příští mikroinstrukce ze dvou a nebo i více adresních polí podle vybrané podmínky (nebo podmínek – vícenásobné větvení).
- **Prostá modifikace** obsahu adresního pole podmínkovým signálem (operace OR, AND, …). Jedná se o velmi rychlý mechanizmus, který ale vykazuje určitá omezení při tvorbě mikroprogramu (cílová mikroinstrukce nemůže ležet na libovolné adrese).

Redukce kapacity řídící paměti

- Rychlost a kapacita paměti jsou protichůdné požadavky. Návrh mikroprogramového řadiče musí vést na co nejmenší nároky co se týká velikosti paměti. Pro dosažení tohoto cíle se používá řada „triků“. Některé redukují celkový počet mikroinstrukcí, potřebných pro implementaci ISA, jiná opatření vedou na zmenšení šířky mikroinstrukčního slova:

- **Redukce celkové délky mikroprogramů**:
  - Prosté sdílení společných sekcí mikroprogramu.
  - Sdílení kódu - použití rutin na mikroprogramové úrovni. Nutný mechanismus pro uložení návratové adresy a návrat do volajícího mikroprogramu (vyžaduje implementaci HW zásobníku pro uložení návratové adresy !!). Hloubka zásobníku může být omezená (4 – 16).
  - Využití HW čítačů cyklů. Čítače dovolují rychlejší a elegantnější implementaci opakování některých sekcí mikroprogramu (s menším počtem mikroinstrukcí).

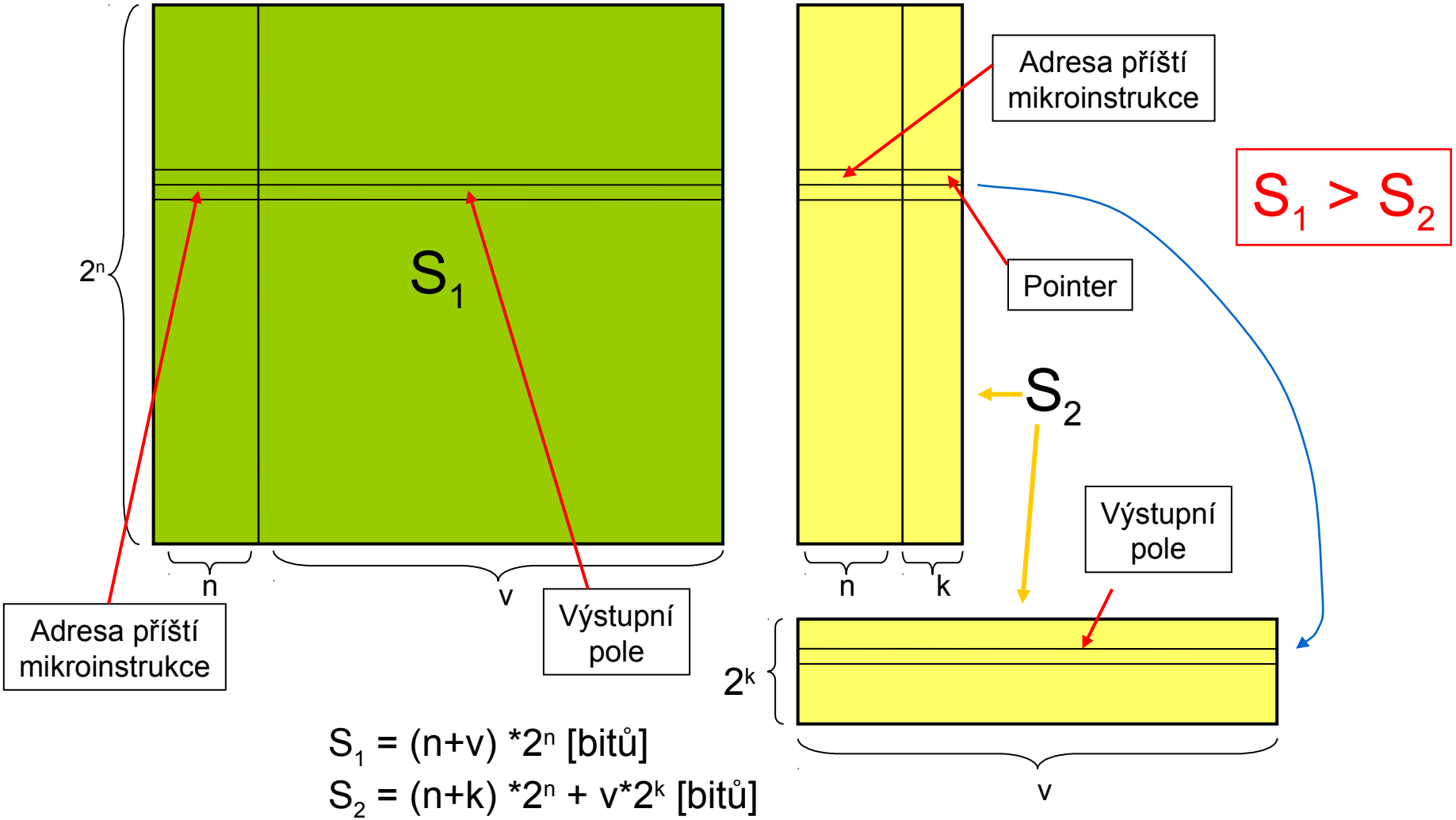
Redukce kapacity řídící paměti: pokračování

- **Redukce celkové délky mikroprogramů** (pokračování):
  - Vícenásobná větvení.
  - Opakování sekce – do splnění podmínky (registr s adresou pro návrat)
- **Redukce šířky mikroinstrukcí:**
  - Malý počet různých mikroinstrukcí, někdy jen jeden typ.
  - Použití mPC – redukuje šíři pole pro příští adresu.
  - Sdílení polí mikroinstrukce. Některé skupiny instrukcí jsou disjunktní z hlediska využívání jednotlivých funkčních bloků procesoru (IO operace, aritmeticko-logické operace, apod.). Jednotlivá výstupní pole mikroinstrukce lze pak sdílet. Jejich význam je interpretován podle typu právě probíhající instrukce.
  - Volbou vhodného kompromisu mezi vertikální a horizontální formou mikroinstrukce.

Redukce kapacity řídící paměti: pokračování

- Nanoprogramování
  - Využívá skutečnosti, že i při značné šířce výstupního pole mikroinstrukce (velký počet řídících signálů) se v celém mikroprogramu objevuje jenom poměrně malý počet odlišných kombinací ve výstupním poli (mnoho kombinací nemá význam).
  - Řícící paměť se skládá ze dvou menších pamětí, ve kterých jsou uloženy:
    - Pointery
    - Vlastní kombinace bitů výstupního pole mikroinstrukcí
  - Čtení výstupního pole mikroinstrukce je dvoustupňové.

Nanoprogramování



$S_1 = (n+v) * 2^n$ [bitů]

$S_2 = (n+k) * 2^n + v * 2^k$ [bitů]

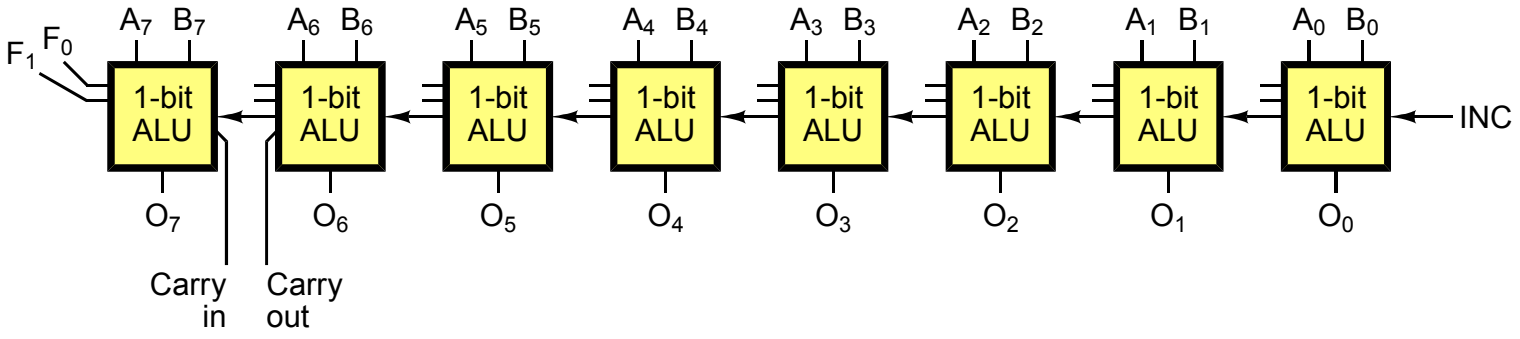
Nanoprogramování: pokračování

- Dosahovaný poměr redukce 2 : 1 až 5 : 1.
- Nevýhodu dvojího čtení bývá možno maskovat, takže se neprojeví zpomalením činnosti řadiče.
- Použito např. u procesoru Motorola MC68000.

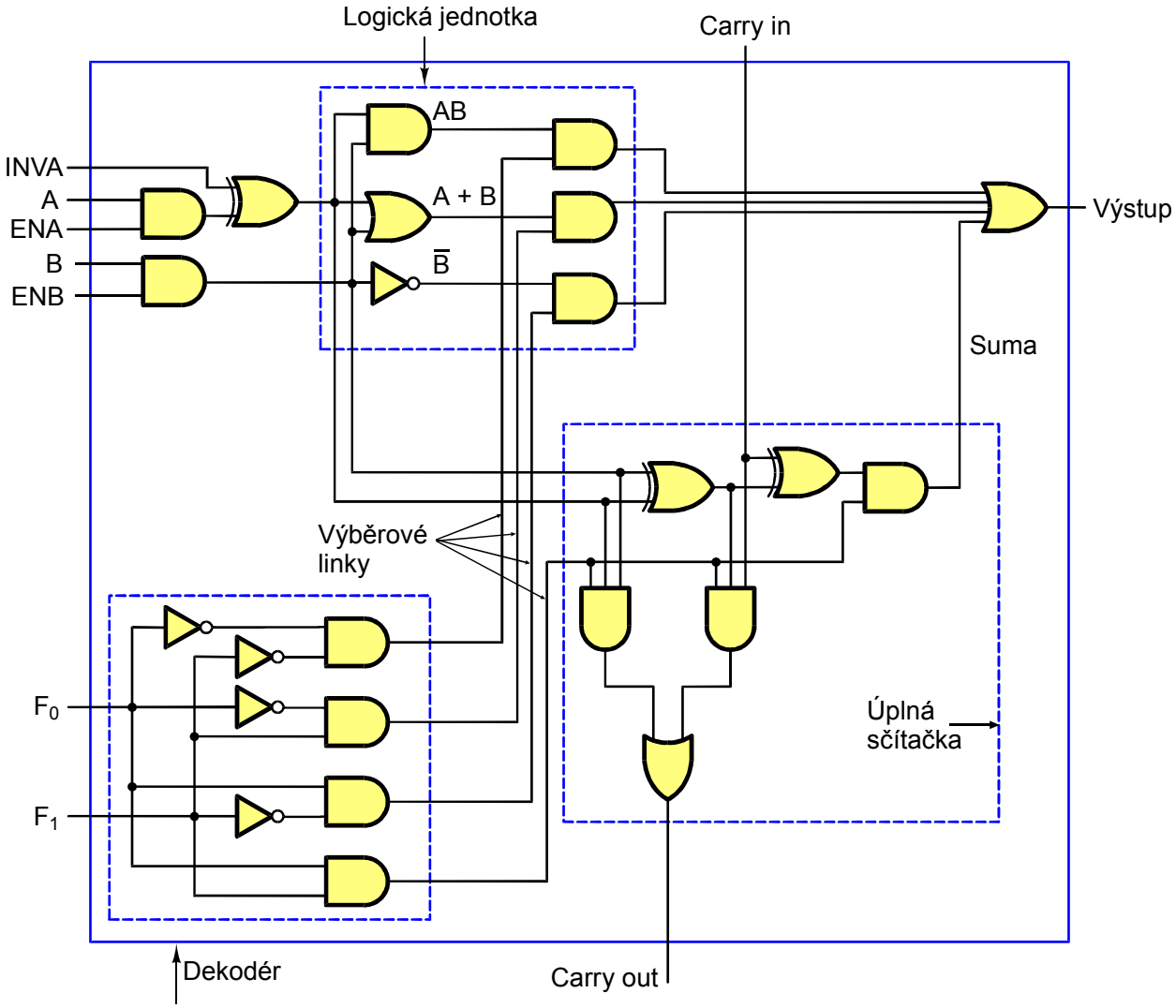
Příklad mikroarchitektury procesoru

- Navržená ISA bude obsahovat jen instrukce, pracující s typem integer.
- Použitá ALU má velmi jednoduchou strukturu a pro jednoduchost jsou vypuštěna veškerá urychlení.

Struktura ALU:



Struktura ALU



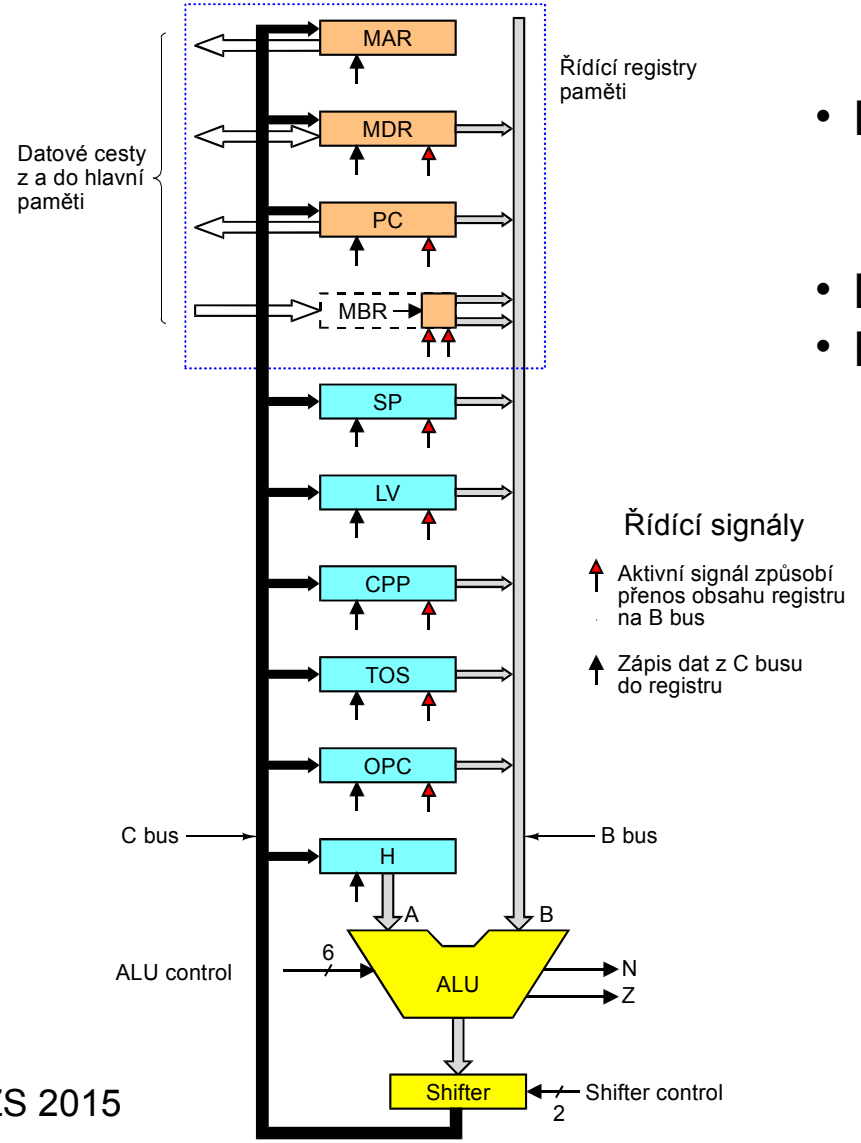
Operace a řízení ALU

F_0	F_1	ENA	ENB	INVA	INC	Funkce
0	1	1	0	0	0	**A**
0	1	0	1	0	0	**B**
0	1	1	0	1	0	**!A**
1	0	1	1	0	0	**!B**
1	1	1	1	0	0	**A + B**
1	1	1	1	0	1	**A + B + 1**
1	1	1	0	0	1	**A + 1**
1	1	0	1	0	1	**B + 1**
1	1	1	1	1	1	**B – A**
1	1	0	1	1	1	**B – 1**
1	1	1	0	1	1	**-A**
0	0	1	1	0	0	**A and B**
0	1	1	1	0	0	**A or C**
0	1	0	0	0	0	**0**
0	1	0	0	0	1	**1**
0	1	0	0	1	0	**- 1**

Řídící signály ALU a funkce, které ALU vykonává.

ALU má čistě kombinační charakter

Datové cesty procesoru



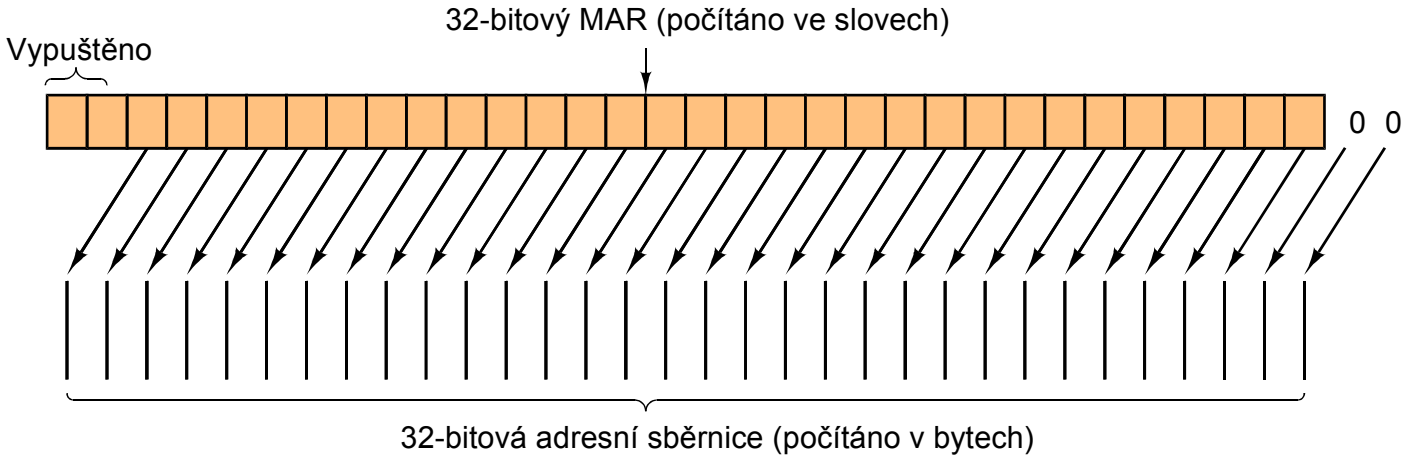
- Dva způsoby komunikace s pamětí:
  - 32 bitů – MAR, MDR (čtení dat)
  - 8 bitů – PC, MBR (čtení instrukcí)
- MAR obsahuje adresy slov
- PC obsahuje adresy bytů

Pozn.
Zobrazena je architektura MIC-1 (Tanenbaum, Structured Computer Organization), určená pro výuku. MIC-1 obsahuje jednoduchou řídící jednotku s mikrokódem (512*36b), 32 registrů, sběrnice, ALU a shifter.

MAR = Memory address reg.
(adresa paměti ze které proc.cte)
MDR = memory data reg. (čtená nebo zap. Hodnota
MBR = memory buffer register (dočasný reg. Hodnoty čtené/zap. z/do paměti)
B-bus = výstup z reg. Vstup do ALU (operand)
C-bus = výstup z ALU/Shifteru, vstup do reg. (výsledek)

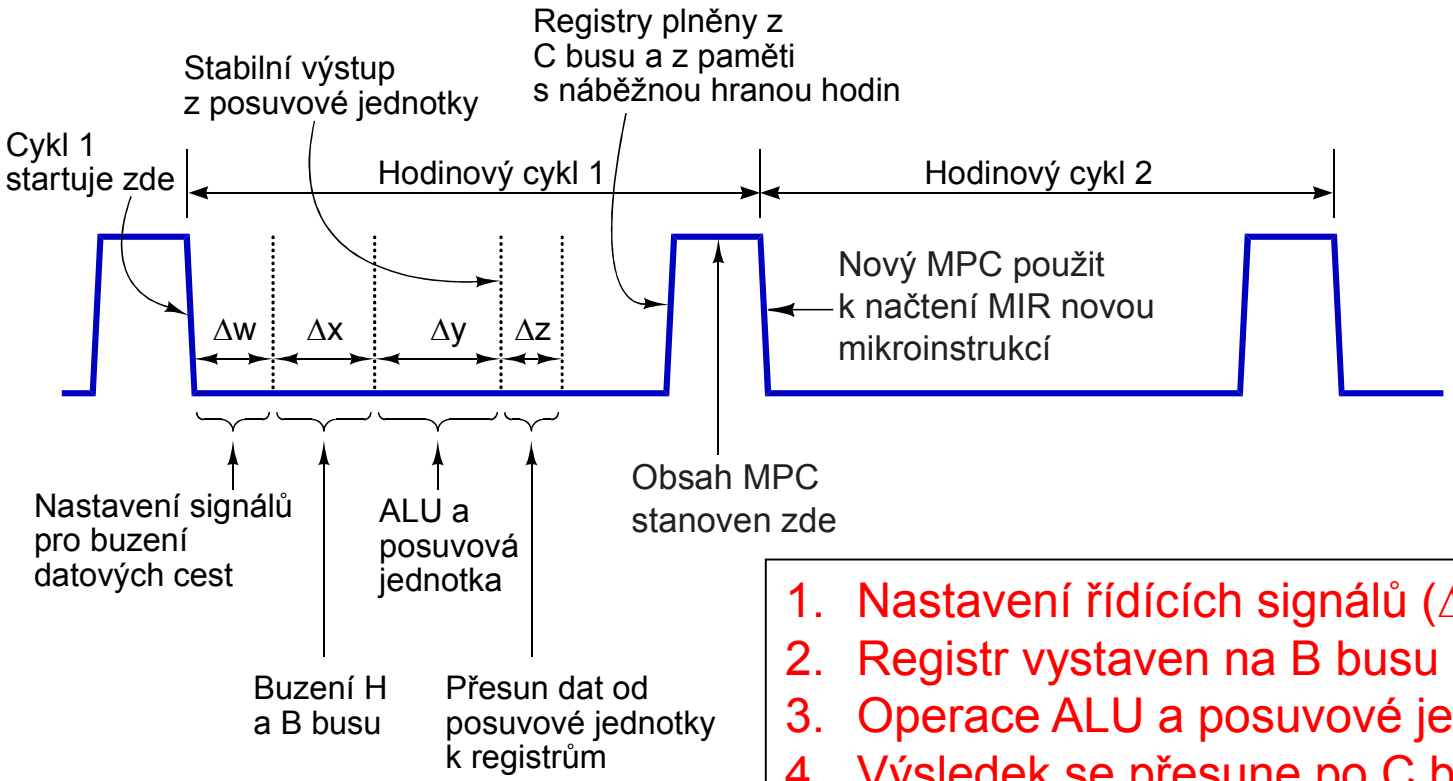
Mapování bitů MAR

Následující obrázek znázorňuje mapování jednotlivých bitů MAR na adresní sběrnici:



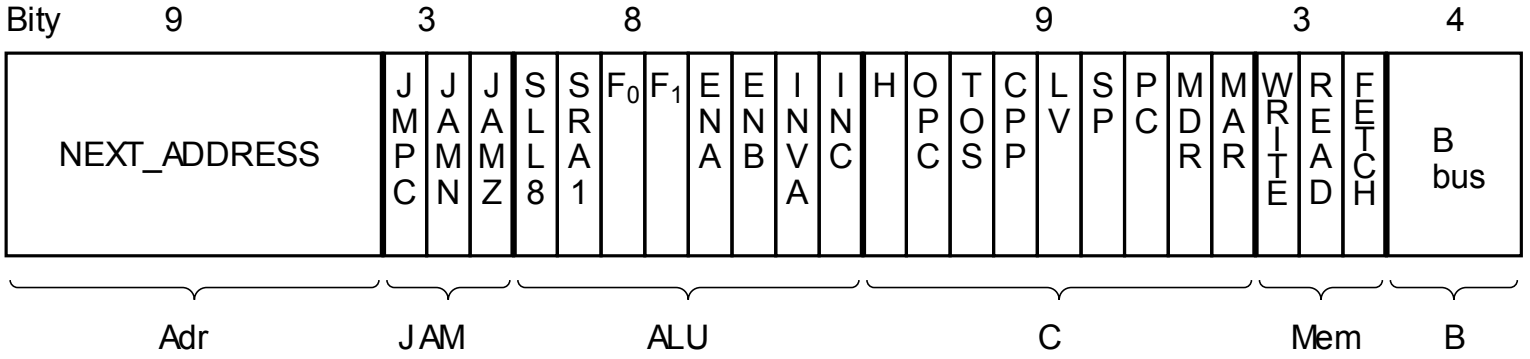
Tento posuv o dva bity vlevo je nutný, aby se obsah MAR vztahoval k bytům a ne ke slovům (4 byty).

Časový diagram cyklu datových cest



1. Nastavení řídících signálů (Δw)
2. Registr vystaven na B busu (Δx)
3. Operace ALU a posuvové jednotky (Δy)
4. Výsledek se přesune po C busu do registru (Δz)

Formát mikroinstrukce MIC-1

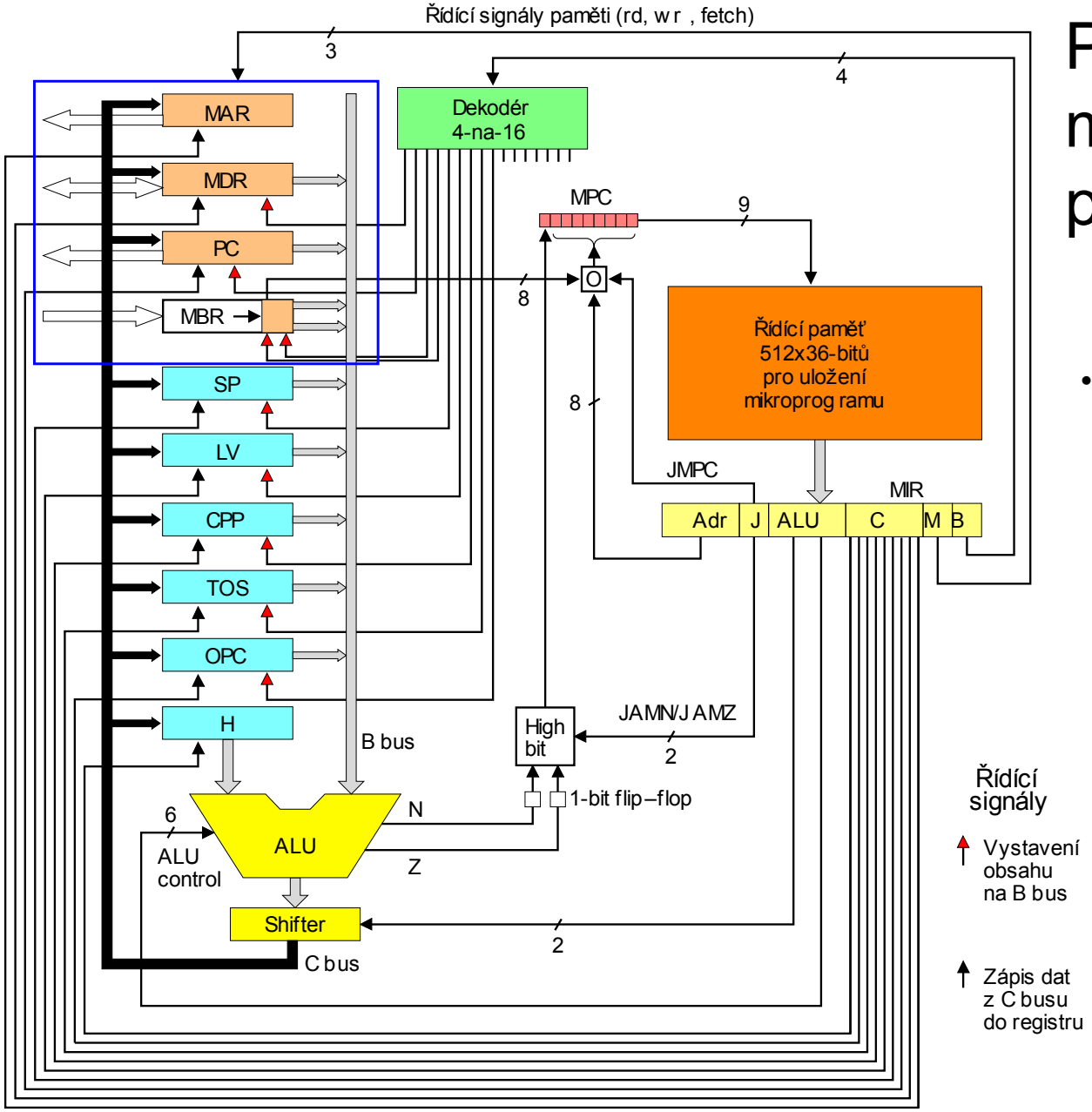


Registry B busu:

0 = MDR	5 = LV
1 = PC	6 = CPP
2 = MBR	7 = TOS
3 = MBR U	8 = OPC
4 = SP	9-15 none

Adr obsahuje adresu potenciální příští mikroinstrukce
JAM určuje způsob výběru příští mikroinstrukce
ALU určuje funkci ALU a posuvové jednotky
C určuje, které registry jsou zapisovány z C-busu
Mem funkce paměti
B zdroj dat pro B bus

Příklad mikroarchitektury procesoru



- Řadič musí v každém cyklu poskytovat informace:
  - stav každého řídícího signálu
  - adresu příští mikroinstrukce

!!! Obrázek neobsahuje jednotku pro načítání instrukcí (IFU), ani instrukční registr.

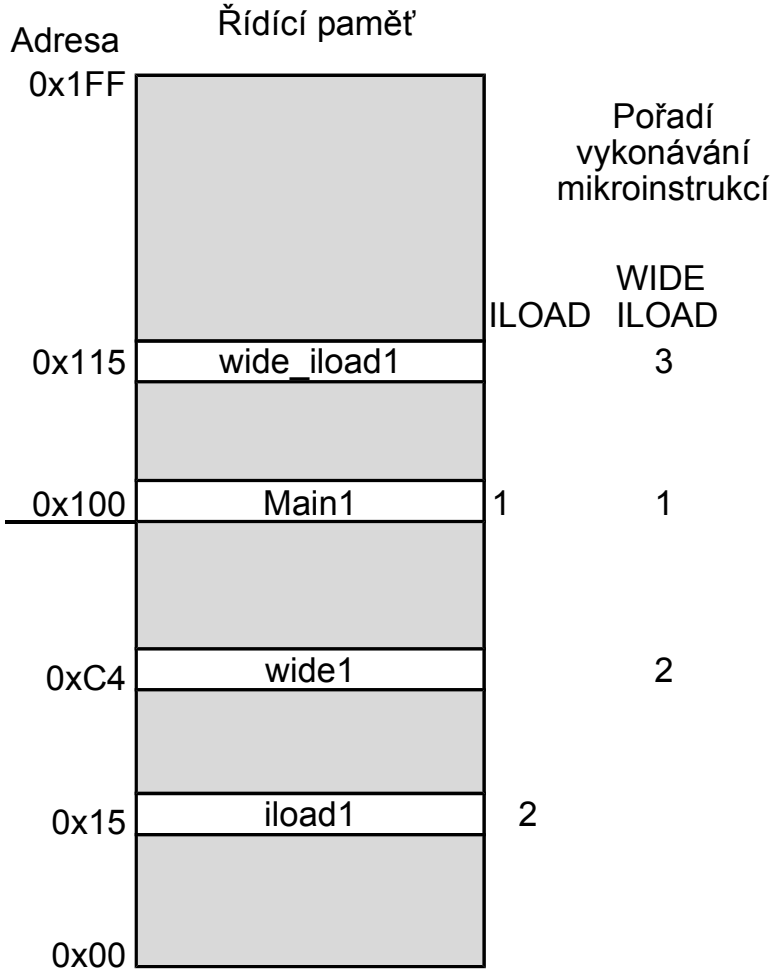
Navrhovaný instrukční soubor IJVM

Hex	Mnemonika	Význam
0x10	BIPUSH byte	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO offset	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ offset	Pop word from stack and branch if it is zero
0x9B	IFLT offset	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMP offset	Pop two words from stack; branch if equal
0x84	IINC varnum const	Add a constant to a local variable
0x15	ILOAD varnum	Push local variable onto stack
0xB6	INVOKEVIRTUAL disp	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
ox36	ISTORE varnum	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W index	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Mikroprogram (část)

Label	Operations	Comments
Main1	PC=PC+1;fetch; goto(MBR)	MBR holds opcode; get next byte; dispatch
nop1	goto Main1	Do nothing
iadd1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iadd2	H = TOS	H = top of stack
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Add top two words; write to top of stack
isub1	MAR = SP = SP - 1; rd	Read in next-to-top word on stack
isub2	H = TOS	H = top of stack
isub3	MDR=TOS=MDR-H; wr; goto Main1	Do subtraction; write to top of stack
iand1	MAR = SP = SP - 1; rd	Read in next-to-top word on stack
iand2	H = TOS	H = top of stack
iand3	MDR = TOS = MDR AND H; wr; goto Main1	Do AND; write to new top of stack
ior1	MAR = SP = SP - 1; rd	Read in next-to-top word on stack
ior2	H = TOS	H = top of stack
ior3	MDR = TOS = MDR OR H; wr; goto Main1	Do OR; write to new top of stack
dup1	MAR = SP = SP + 1	Increment SP and copy to MAR
dup2	MDR = TOS; wr; goto Main1	Write new stack word
pop1	MAR = SP = SP - 1; rd	Read in next-to-top word on stack
pop2		Wait for new TOS to be read from memory
pop3	TOS = MDR; goto Main1	Copy new word toTOS
swap1	MAR = SP - 1; rd	Set MAR to SP - 1; read 2nd word from stack
swap2	MAR = SP	Set MAR to top word
swap3	H = MDR; wr	Save TOS in H; write 2nd word to top of stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP - 1; wr	Set MAR to SP - 1; write as 2nd word on stack
swap6	TOS = H; goto Main1	UpdateTOS
bipush1	SP = MAR = SP + 1	MBR = the byte to push onto stack

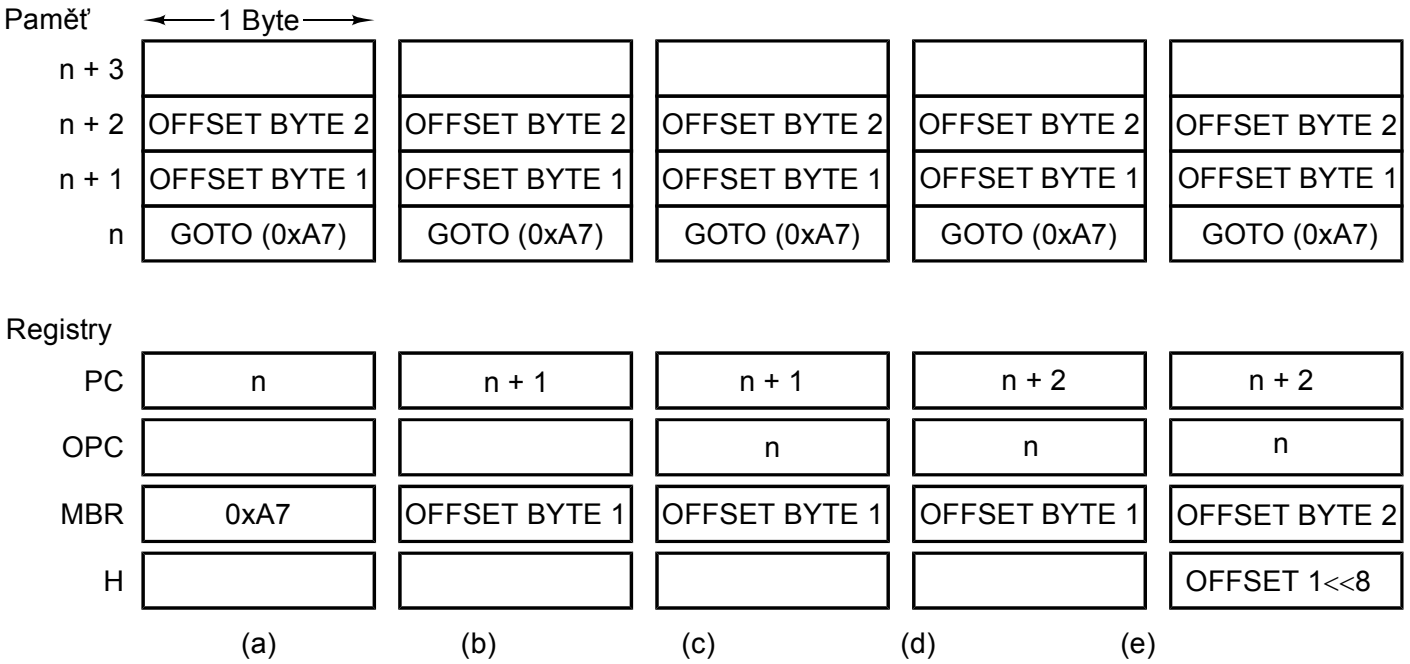
Mikroinstrukce ILOAD a WIDE ILOAD



Počáteční posloupnost mikroinstrukcí při zpracování ILOAD a WIDE ILOAD

WIDE je prefix.

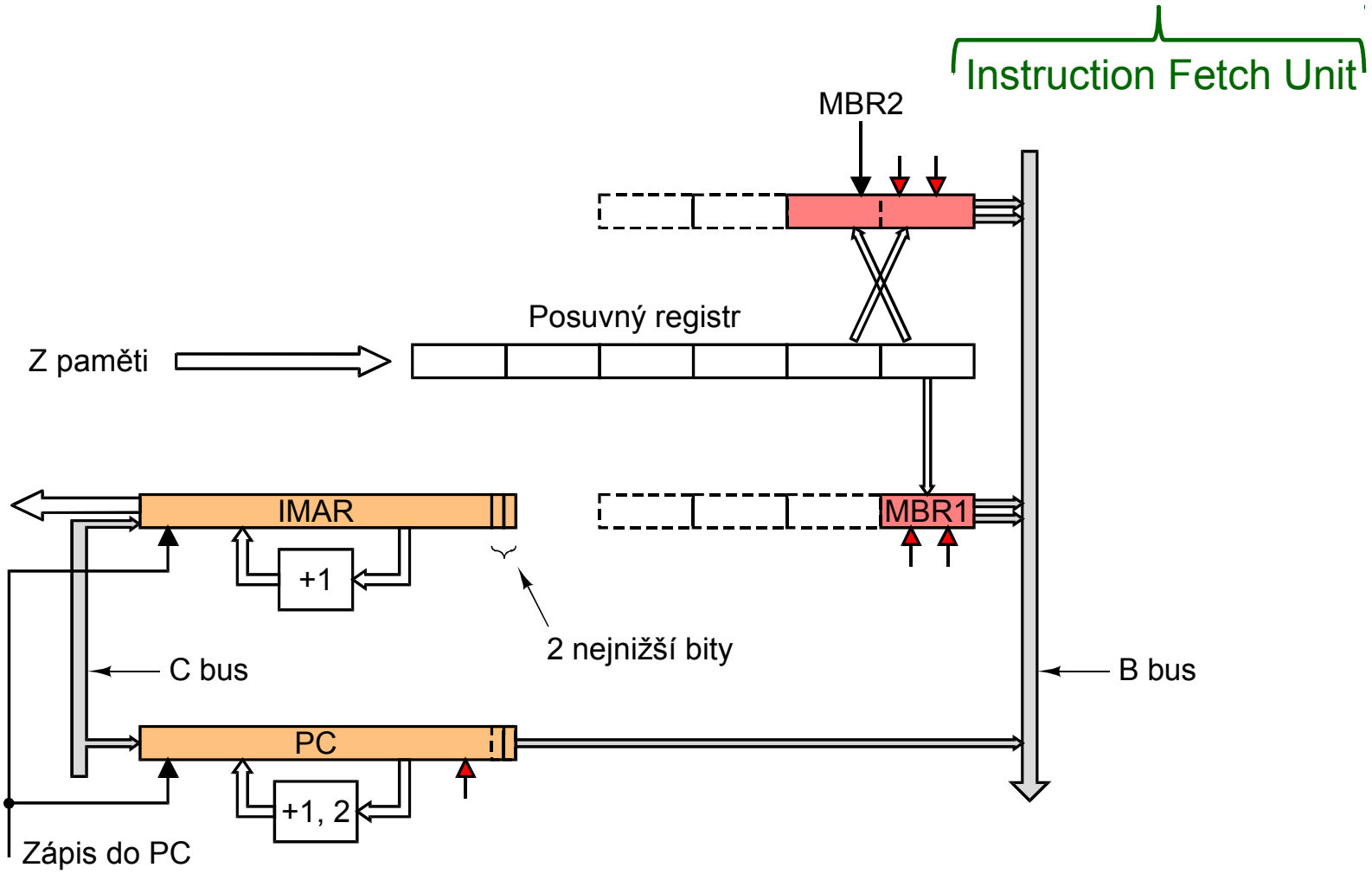
Některé mikroinstrukce



Situace při zahájení některých mikroinstrukcí:

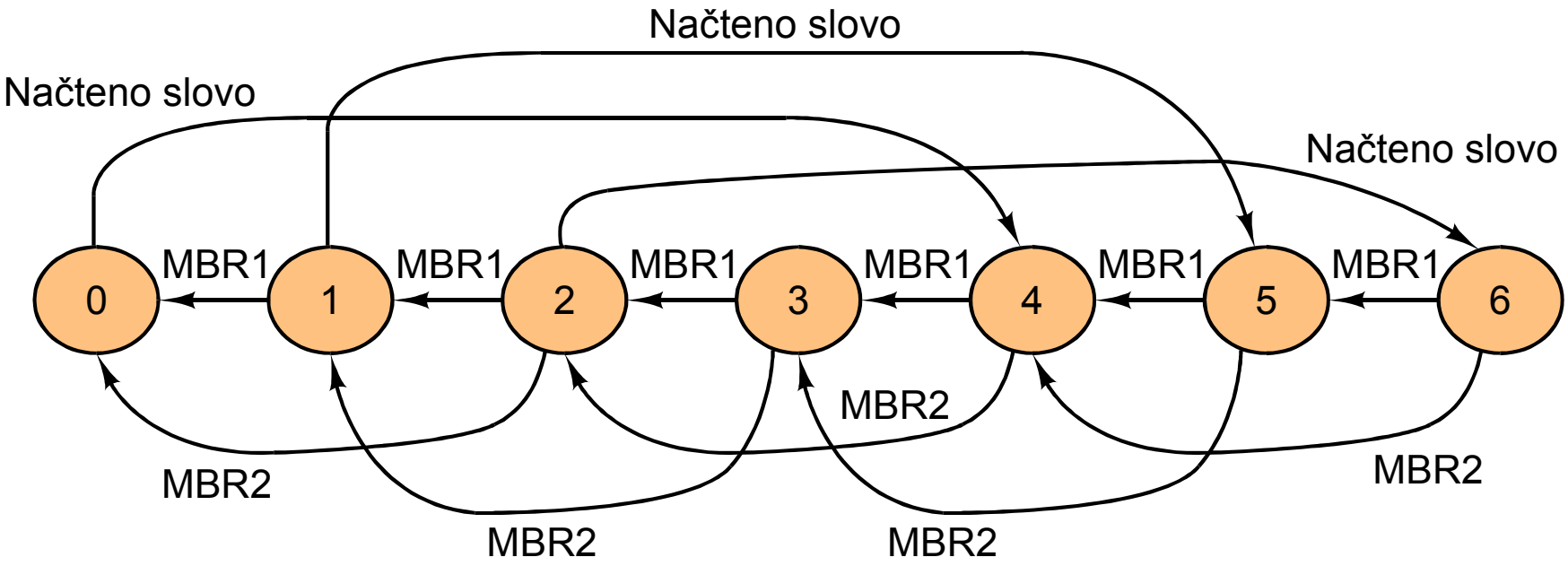
a) Main1
b) goto1
c) goto2
d) goto3
e) goto4

Jednotka načtení instrukce (IFU)



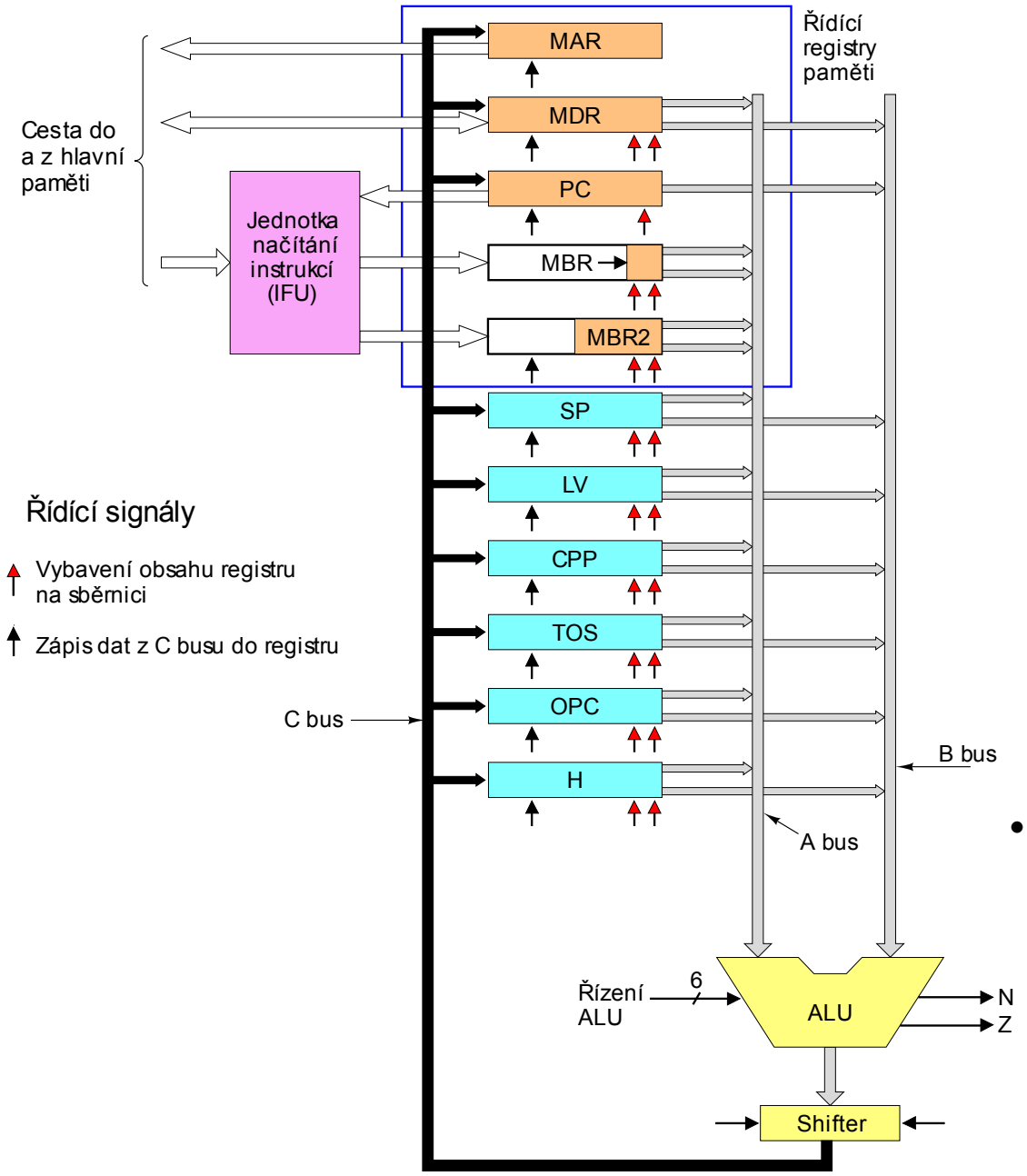
Automat pro implementaci IFU

Instruction Fetch Unit



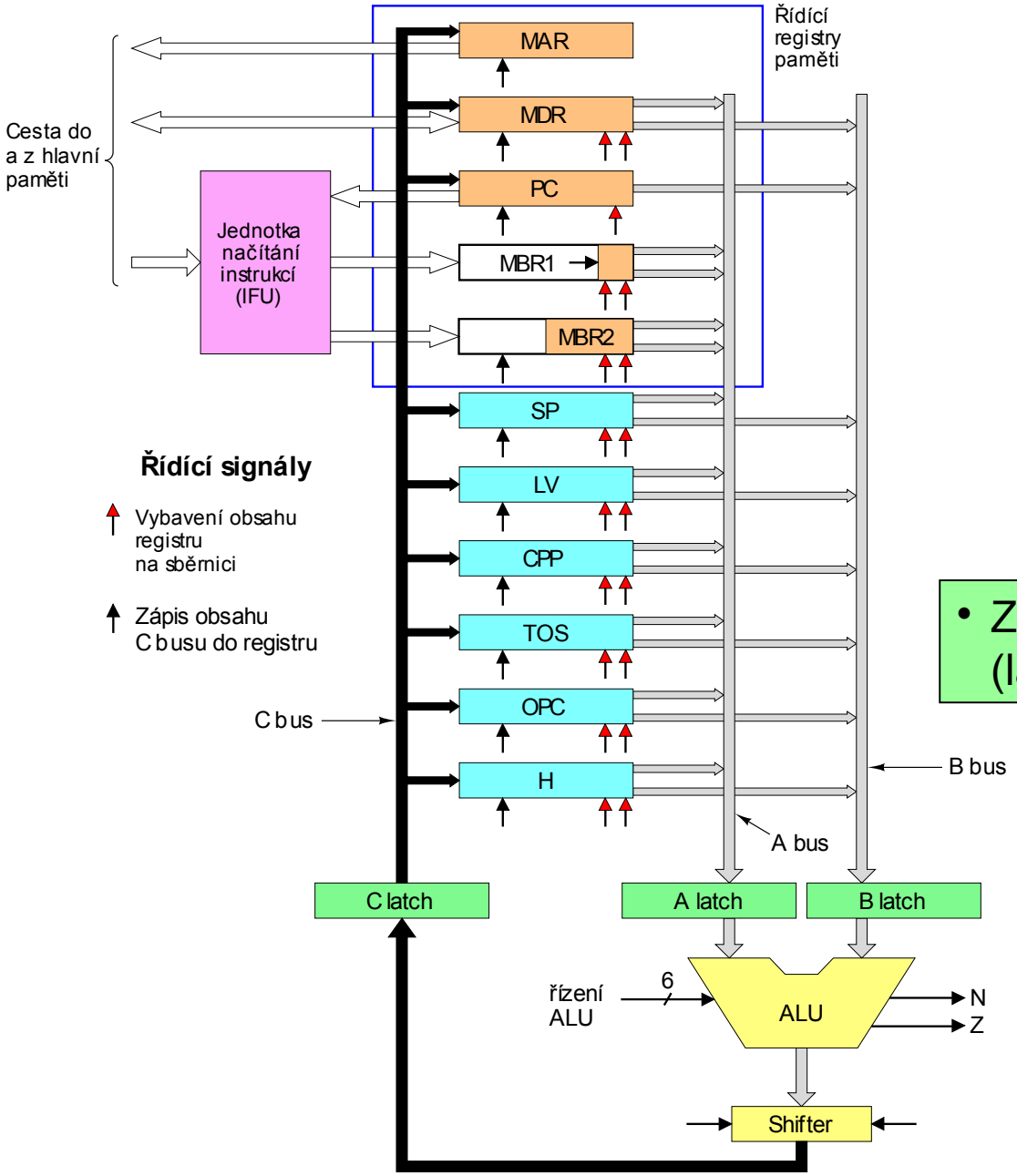
MBR1 nastane, je li načten MBR1
MBR2 nastane, je-li načten MBR2

Datové cesty procesoru – verze 2



- Do operace mohou vstoupit i jiné dvojice registrů než jen H + x.
  - Snížení počtu mikroinstrukcí.
  - Zkrácení doby instrukcí

Datové cesty procesoru – verze 3



- Zařazení oddělovacích registrů (latch) umožní pipelinning.

Příklad - ISA MIPS32

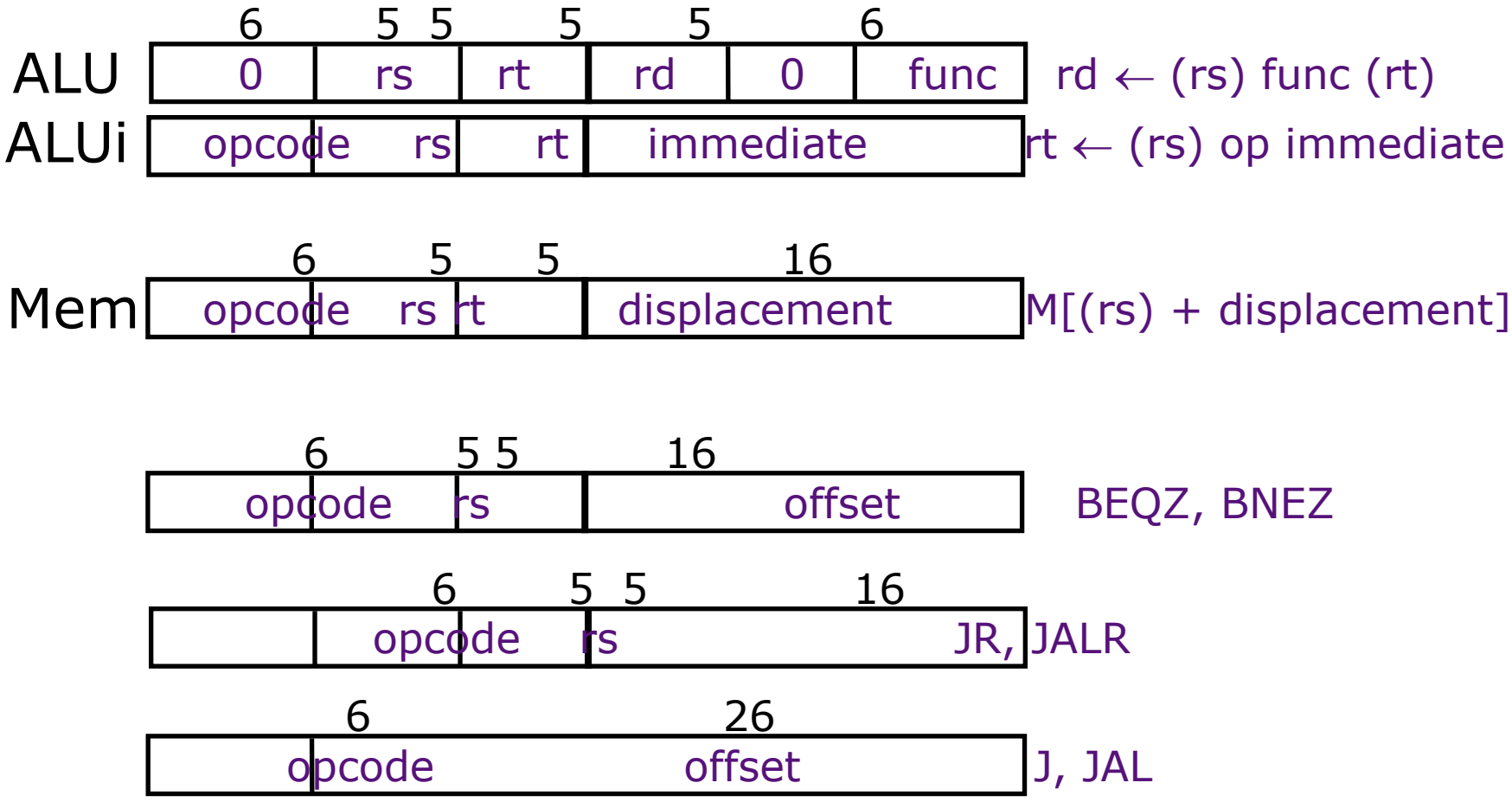
- Stav procesoru
  32 32-bitových GPR, R0 obsahuje 0
  16 double-precision/32 single-precision FPR
  FP stavový registr, používaný pro FP porovnání & výjimky
  PC čítač adres instrukcí
  další speciální registry

- Datové typy
  8-bitů byte, 16-bitů půlslovo
  32-bitů slovo pro integer
  32-bitů slovo pro single precision floating point
  64-bitů slovo pro double precision floating point

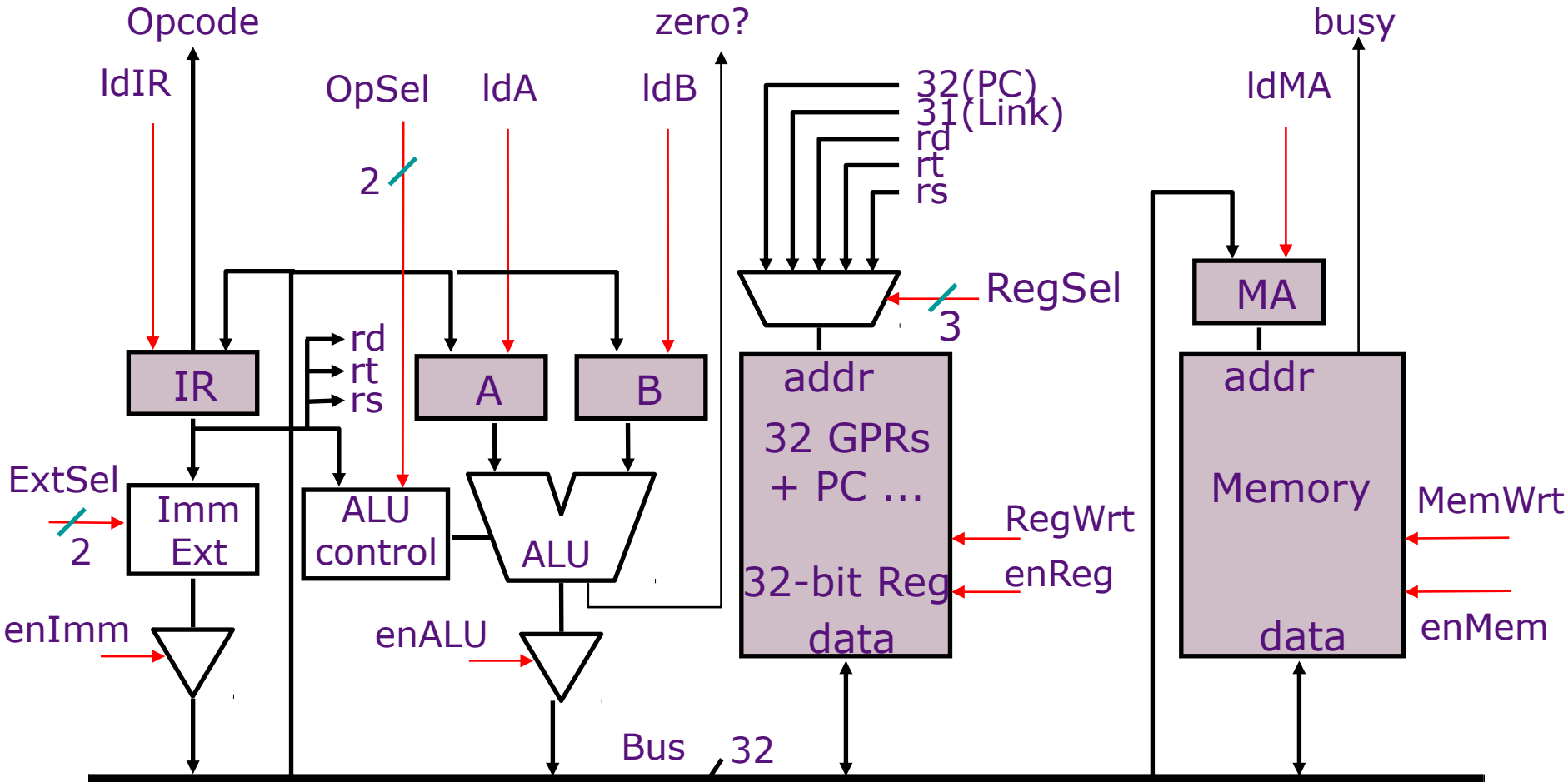
- Instrukční soubor stylu Load/Store
  datové adresní módy- přímý operand & indexace
  adresní módy větvení- PC relativní & registr indirekt
  bytově adresovaná paměť - big-endian mód

Všechny instrukce jsou dlouhé 32 bitů

Instrukční formáty MIPS



Datové cesty MIPS: sběrnicová koncepce

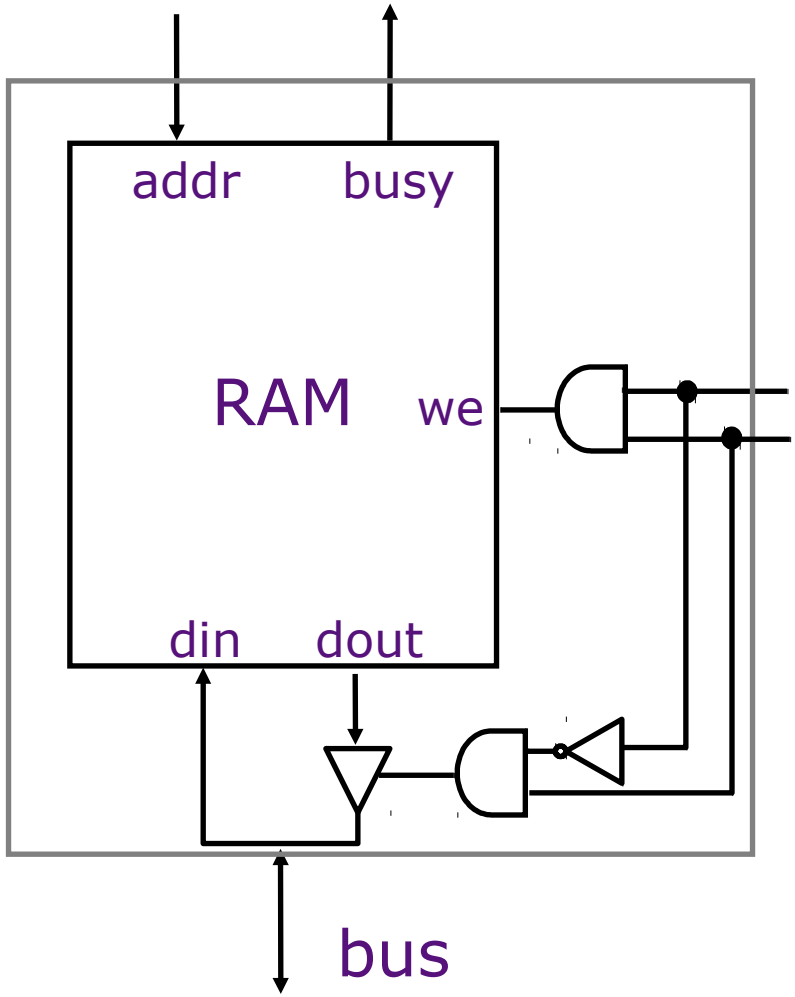


Mikroinstrukce: přenos registr-registr (17 řídících signálů)

MA ← PC *znamená* RegSel = PC; enReg=yes; ldMA= yes

B ← Reg[rt] *znamená* RegSel = rt; enReg=yes; ldB = yes

Modul paměti



Write(1)/Read(0)

Enable

Předpoklad:

Paměť pracuje nezávisle a je pomalejší ve srovnání s přenosy mezi registry (více hodinových taktů CPU na jeden přístup do paměti)

Provedení instrukce

Provedení instrukce MIPS zahrnuje:

1. načtení instrukce
2. dekódování a naplnění registru
3. operace ALU
4. operace paměti (některé instrukce)
5. zpětný zápis výsledku do registrového souboru (v některých případech)

\+ výpočet adresy *příští instrukce*

Části mikroprogramu

instr fetch: MA ← PC
A ← PC
IR ← Memory
PC ← A + 4
dispatch on OPcode

lze chápat jako makro

ALU: A ← Reg[rs]
B ← Reg[rt]
Reg[rd] ← func(A,B)
do instruction fetch

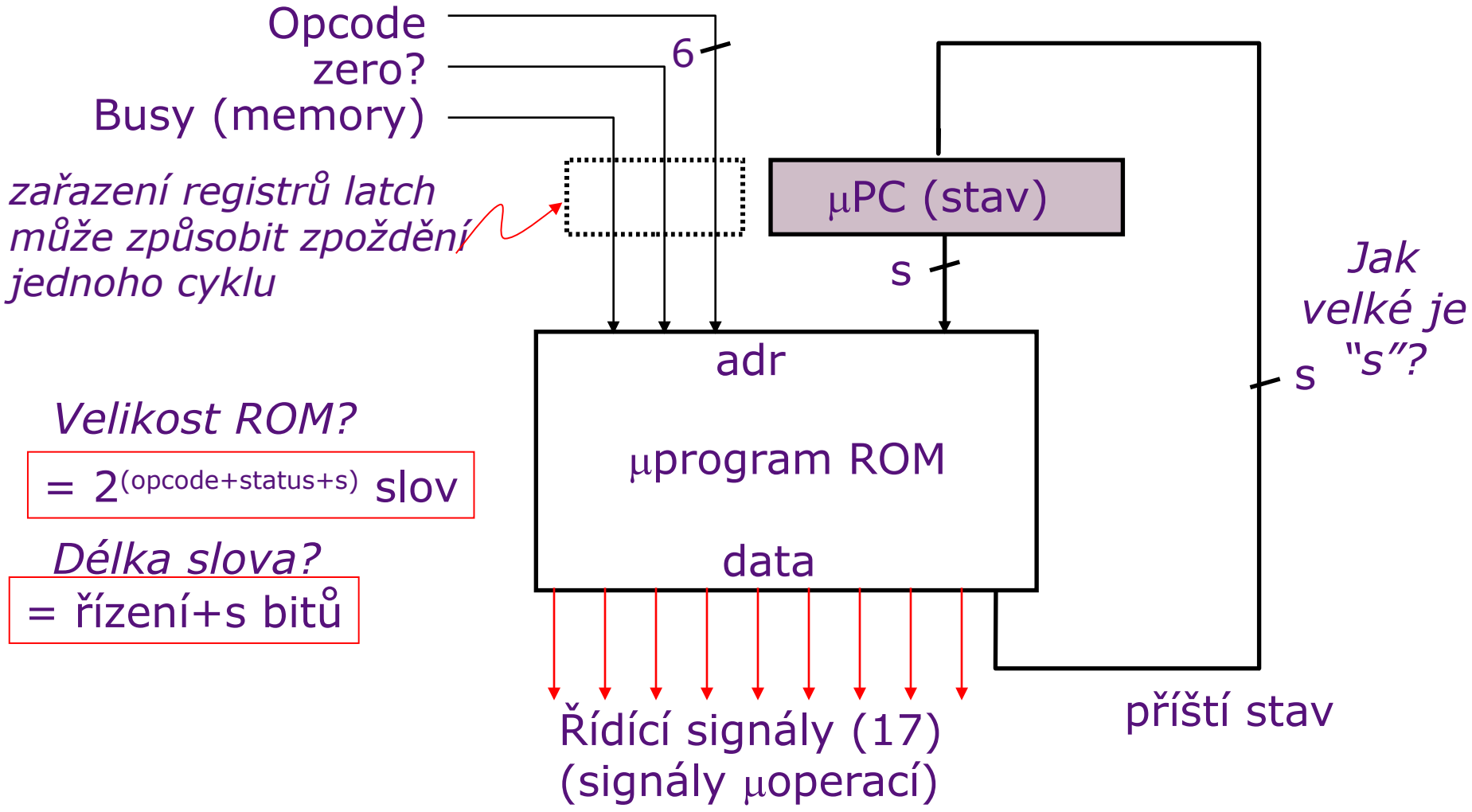
ALUi: A ← Reg[rs]
B ← Imm *expanze znaménka ...*
Reg[rt] ← Opcode(A,B)
do instruction fetch

Části mikroprogramu *(pokrač.)*

```
LW:     A ← Reg[rs]
        B ← Imm
        MA ← A + B
        Reg[rt] ← Memory
        do instruction fetch

J:      A ← PC
        B ← IR
        PC ← JumpTarg(A,B)
        do instruction fetch

beqz:   A ← Reg[rs]
        If zero?(A) then go to bz-taken
        do instruction fetch

bz-taken: A ← PC
        B ← Imm << 2
        PC ← A + B
        do instruction fetch
```

JumpTarg(A,B) = {A[31:28],B[25:0],00}

Mikrořadič MIPS: *prvý pokus*



Mikroprogram v ROM *pracovní verze*

Stav	Op	zero?	busy	Mikrooperace	Příští stav
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
$fetch_3$	*	*	*	PC ← A + 4	?
$fetch_3$	ALU	*	*	PC ← A + 4	ALU_0
ALU_0	*	*	*	A ← Reg[rs]	ALU_1
ALU_1	*	*	*	B ← Reg[rt]	ALU_2
ALU_2	*	*	*	Reg[rd] ← func(A,B)	$fetch_0$

Mikroprogram v ROM

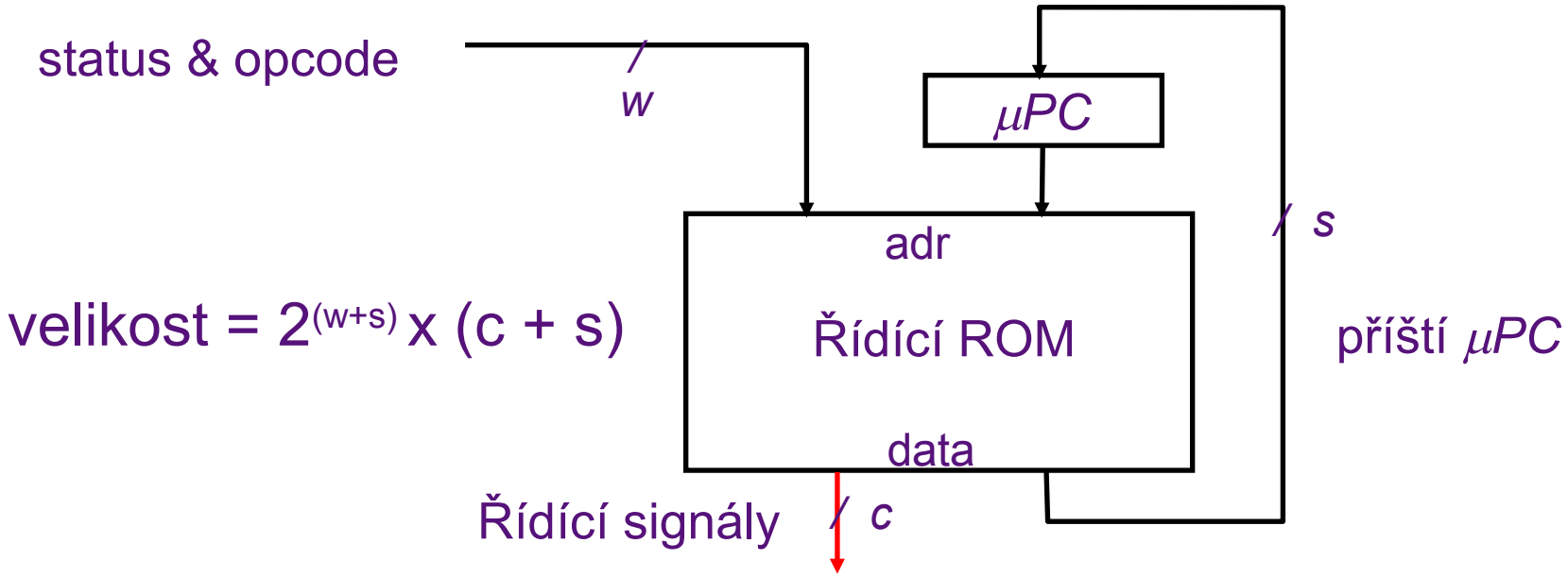
Stav	Op	zero?	busy	Mikrooperace	Příští stav
$fetch_0$	*	*	*	MA ← PC	$fetch_1$
$fetch_1$	*	*	yes	....	$fetch_1$
$fetch_1$	*	*	no	IR ← Memory	$fetch_2$
$fetch_2$	*	*	*	A ← PC	$fetch_3$
$fetch_3$	ALU	*	*	PC ← A + 4	ALU_0
$fetch_3$	ALUi	*	*	PC ← A + 4	$ALUi_0$
$fetch_3$	LW	*	*	PC ← A + 4	LW_0
$fetch_3$	SW	*	*	PC ← A + 4	SW_0
$fetch_3$	J	*	*	PC ← A + 4	J_0
$fetch_3$	JAL	*	*	PC ← A + 4	JAL_0
$fetch_3$	JR	*	*	PC ← A + 4	JR_0
$fetch_3$	JALR	*	*	PC ← A + 4	ALR_0
$fetch_3$	beqz	*	*	PC ← A + 4	$beqz_0$
...					
ALU_0	*	*	*	A ← Reg[rs]	ALU_1
ALU_1	*	*	*	B ← Reg[rt]	ALU_2
ALU_2	*	*	*	Reg[rd] ← func(A,B)	$fetch_0$

Mikroprogram v ROM: *pokračování*

Stav	Op	zero?	busy	Mikrooperace	Příští stav
$ALUi_0$	*	*	*	A ← Reg[rs]	$ALUi_1$
$ALUi_1$	sExt	*	*	B ← $sExt_{16}$(Imm)	$ALUi_2$
$ALUi_1$	uExt	*	*	B ← $uExt_{16}$(Imm)	$ALUi_2$
$ALUi_2$	*	*	*	Reg[rd]← Op(A,B)	$fetch_0$
...					
J_0	*	*	*	A ← PC	J_1
J_1	*	*	*	B ← IR	J_2
J_2	*	*	*	PC ← JumpTarg(A,B)	$fetch_0$
...					
$beqz_0$	*	*	*	A ← Reg[rs]	$beqz_1$
$beqz_1$	*	yes	*	A ← PC	$beqz_2$
$beqz_1$	*	no	*	....	$fetch_0$
$beqz_2$	*	*	*	B ← $sExt_{16}$(Imm)	$beqz_3$
$beqz_3$	*	*	*	PC ← A+B	$fetch_0$
...					

JumpTarg(A,B) = {A[31:28],B[25:0],00}

Velikost řídící paměti



velikost = $2^{(w+s)}$ x (c + s)

MIPS: w = 6+2 c = 17 s = ?

kroků na opcode = 4 až 6 + fetch-sequence

stavů ≈ (4 kroky na op-group) x op-groups + společné sekvence

= 4 x 8 + 10 stavů = 42 stavů ⇒ s = 6

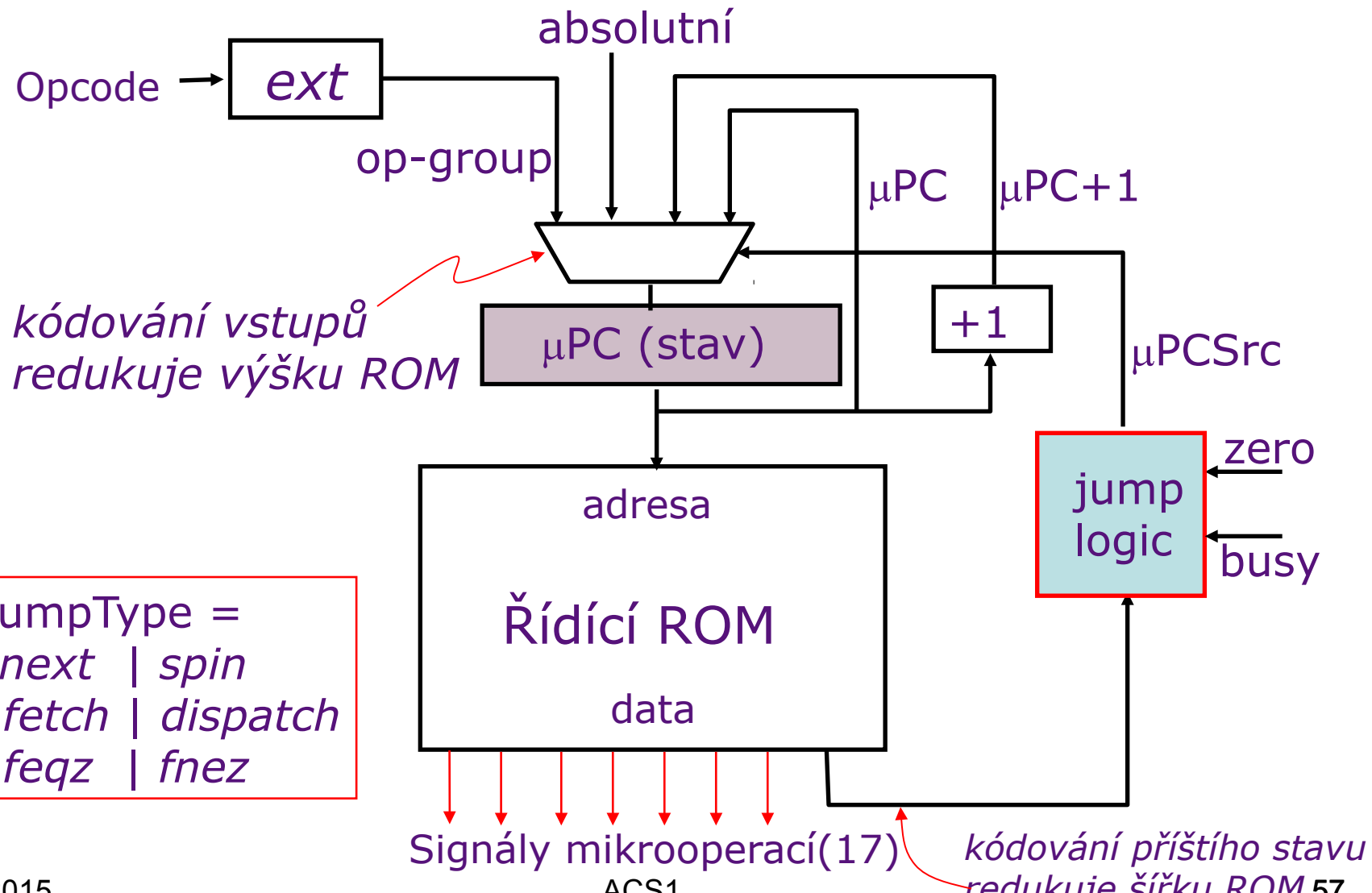
Řídící ROM = $2^{(8+6)}$ x 23 bitů ≈ 48 Kbytů

Redukce velikosti řídící paměti

Řídící paměť musí být rychlá $\Rightarrow$ *drahá*

- Redukovat výšku ROM (= #adresních bitů)
  - *redukovat #vstupů další externí logikou*
    každý vstupní bit zdvojnásobuje velikost řídící paměti
  - *redukovat stavy seskupováním op. kódů*
    nalézt společné sekvence akcí
  - *kondenzovat vstupní stavové bity*
    kombinovat všechny výjimky do jediné, tzn., výjimka nastala/výjimka nenastala
    - Redukovat šířku ROM
  - *omezit dekódování příštího stavu*
    Next, Dispatch on opcode, Wait (na paměť), ...
  - *kódovat řídící signály (vertikální mikroinstrukce)*

Mikrořadič MIPS: Verze 2



Logika skoků

μPCSrc = *Case* μJumpTypes

Způsoby tvorby adresy příští μinstrukce:

next	⇒	μPC+1
spin	⇒	if (busy) then μPC else μPC+1
fetch	⇒	absolute
dispatch	⇒	op-group
feqz	⇒	if (zero) then absolute else μPC+1
fnez	⇒	if (zero) then μPC+1 else absolute

Název metody

Načtení instrukce & ALU: *MIPS-řadič-2*

Stav	Mikrooperace	Příští stav
$fetch_0$	MA ← PC	next
$fetch_1$	IR ← Memory	spin
$fetch_2$	A ← PC	next
$fetch_3$	PC ← A + 4	dispatch
...		
ALU_0	A ← Reg[rs]	next
ALU_1	B ← Reg[rt]	next
ALU_2	Reg[rd]←func(A,B)	fetch
$ALUi_0$	A ← Reg[rs]	next
$ALUi_1$	B ← $sExt_{16}$(Imm)	next
$ALUi_2$	Reg[rd]← Op(A,B)	fetch

Load & Store: *MIPS-řadič-2*

Stav	Mikrooperace	Příští stav
LW_0	A ← Reg[rs]	next
LW_1	B ← $sExt_{16}$(Imm)	next
LW_2	MA ← A+B	next
LW_3	Reg[rt] ← Memory	spin
LW_4	fetch	
SW_0	A ← Reg[rs]	next
SW_1	B ← $sExt_{16}$(Imm)	next
SW_2	MA ← A+B	next
SW_3	Memory ← Reg[rt]	spin
SW_4	fetch	

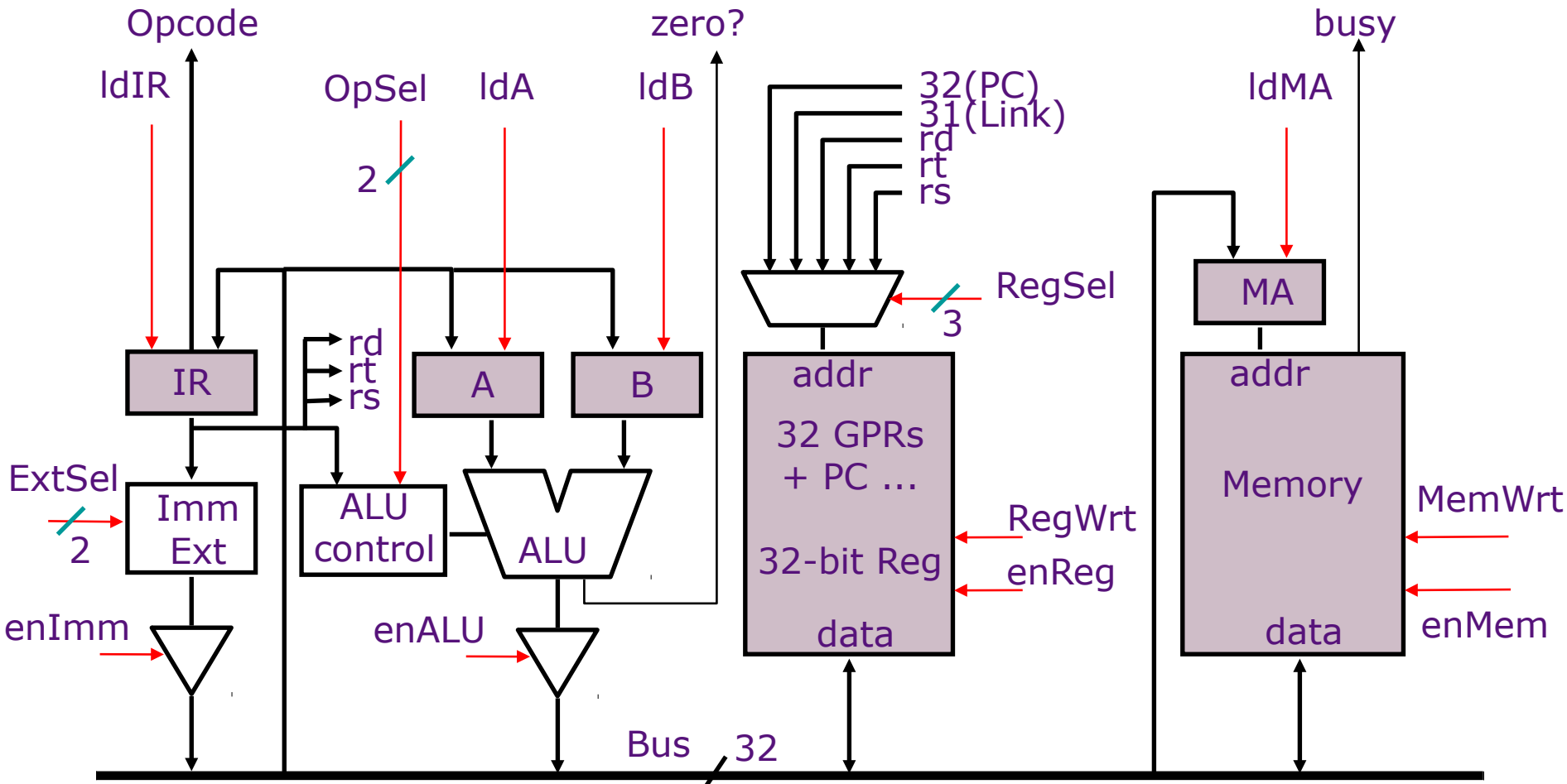
Větvení: *MIPS-řadič-2*

Stav	Mikrooperace	Příští stav
$BEQZ_0$	$A \leftarrow Reg[rs]$	next
$BEQZ_1$	fnez	
$BEQZ_2$	$A \leftarrow PC$	next
$BEQZ_3$	$B \leftarrow sExt_{16}(Imm<<2)$	next
$BEQZ_4$	$PC \leftarrow A+B$	fetch
$BNEZ_0$	$A \leftarrow Reg[rs]$	next
$BNEZ_1$	feqz	
$BNEZ_2$	$A \leftarrow PC$	next
$BNEZ_3$	$B \leftarrow sExt_{16}(Imm<<2)$	next
$BNEZ_4$	$PC \leftarrow A+B$	fetch

Skoky: *MIPS-řadič-2*

State	Control points	next-state
J_0	A ← PC	next
J_1	B ← IR	next
J_2	PC ← JumpTarg(A,B)	fetch
JR_0	A ← Reg[rs]	next
JR_1	PC ← A	fetch
JAL_0	A ← PC	next
JAL_1	Reg[31] ← A	next
JAL_2	B ← IR	next
JAL_3	PC ← JumpTarg(A,B)	fetch
$JALR_0$	A ← PC	next
$JALR_1$	B ← Reg[rs]	next
$JALR_2$	Reg[31] ← A	next
$JALR_3$	PC ← B	fetch

Implementace složitých instrukcí



rd ← M[(rs)] op (rt) *Reg-Memory-src ALU op*

M[(rd)] ← (rs) op (rt) *Reg-Memory-dst ALU op*

M[(rd)] ← M[(rs)] op M[(rt)] *Mem-Mem ALU op*

Instrukce Mem-Mem ALU: *MIPS-řadič-2*

Mem-Mem ALU op M[(rd)] ← M[(rs)] op M[(rt)]

$ALUMM_0$	MA ← Reg[rs]	next
$ALUMM_1$	A ← Memory	spin
$ALUMM_2$	MA ← Reg[rt]	next
$ALUMM_3$	B ← Memory	spin
$ALUMM_4$	MA ←Reg[rd]	next
$ALUMM_5$	Memory ← func(A,B)	spin
$ALUMM_6$		fetch

Složité instrukce obvykle nevyžadují modifikace datových cest při implementaci pomocí mikroprogramu

-- vyžadují pouze prostor pro řídící program

Přímá HW implementace těchto instrukcí je obvykle bez modifikace datových cest obtížná

Výkonové důsledky

Mikroprogramové řízení

$\Rightarrow$ větší počet cyklů na jednu instrukci

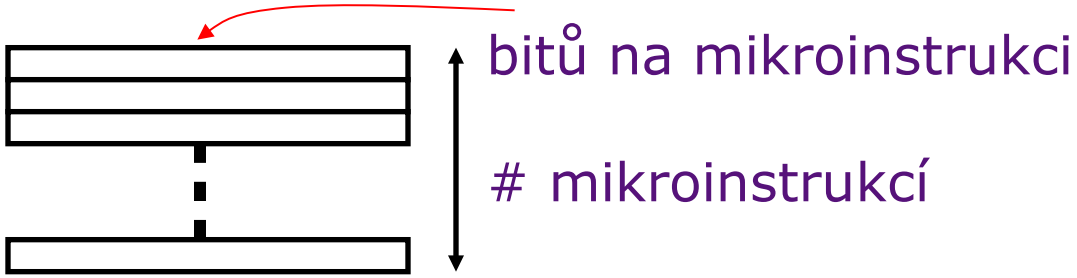
Doba cyklu ?

$t_C > \max(t_{reg\text{-}reg}, t_{ALU}, t_{\mu ROM})$

Předpokládejme $10 * t_{\mu ROM} < t_{RAM}$

Dobrého výkonu, relativně k jednocyklové HW implementaci, lze dosáhnout i při CPI kolem 10

Horizontální vs vertikální μkód



- Horizontální μkód používá široké μinstrukce
  - Paralelní operace během jedné μinstrukce
  - Méně kroků na provedení jedné makroinstrukce
  - Řídké kódování ⇒ větší počet bitů
- Vertikální forma μinstrukce – μkód má užší μinstrukce
  - Typicky operace jednoduché datové cesty na jednu μinstrukci
    - vyhrazené μinstrukce pro větvení
  - Více kroků na provedení jedné makroinstrukce
  - Kompaktnější ⇒ méně bitů
- Nanoprogramování
  - Kombinuje lepší vlastnosti horizontálního a vertikálního μkódu

Nanoprogramování

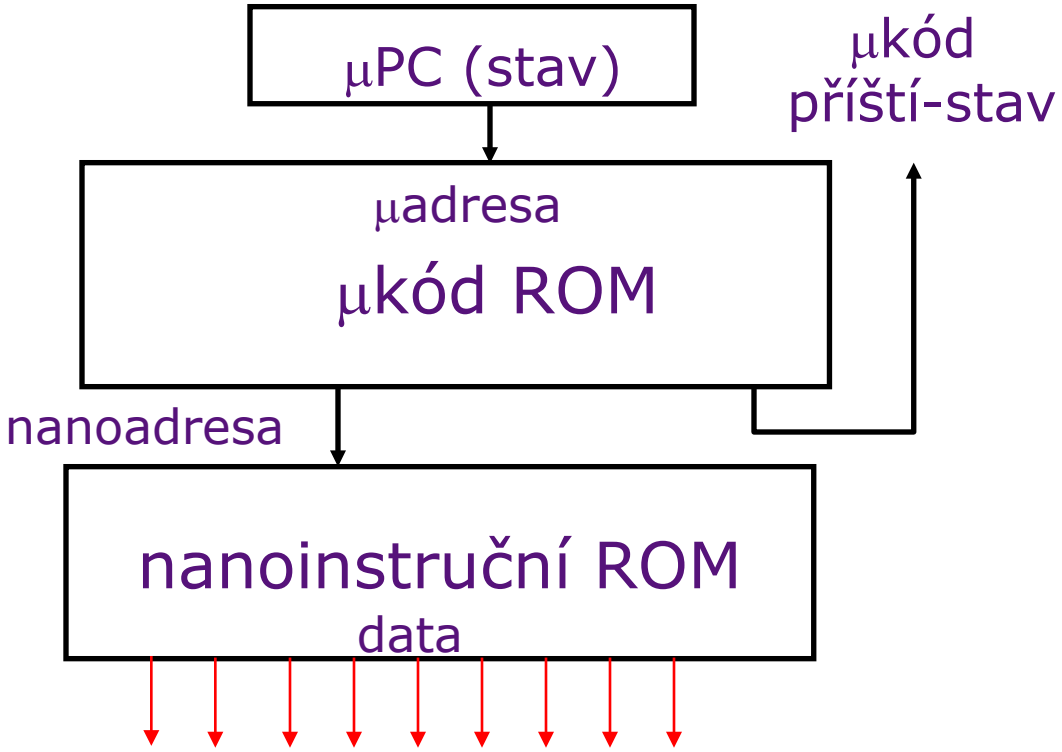
Využívá opakující se kombinace řídících signálů v μkódu, např.,

ALU_0 A ← Reg[rs]

...

$ALUi_0$ A ← Reg[rs]

...



- MC68000 měla 17-bitový μkód obsahující buď 10-bitový μjump nebo 9-bitový nanoinstrukční pointer
  - Nanoinstrukce byly 68 bitů široké, po dekódování dávaly 196 řídících signálů

Mikroprogramování IBM 360

	M30	M40	M50	M65
Šířka dat (bit)	8	16	32	64
Šířka μinstrukce (bit)	50	52	85	87
Velikost μkódu (K μinst)	4	4	2.75	2.75
Technologie řídící paměti	CCROS	TCROS	BCROS	BCROS
Cykl řídící paměti (ns)	750	625	500	200
Cykl hlavní paměti (ns)	1500	2500	2000	750
Pronájem ($K/měsíc)	4	7	15	35

Jen nejrychlejší modely (75 a 95) měly hardwarové řízení (nikoliv μprogramové)

Emulace mikrokódem

- IBM při zavádění řady 360 původně špatně odhadla význam softwarové kompatibility se staršími modely
- Honeywell ukradl některé zákazníky IBM 1401 tak, že nabídl translační software (“Liberator”) pro řadu Honeywell H200
- IBM odpověděla vytvořením přídavného mikrokódu pro řadu 360 tak, že mohla emulovat IBM 1401 ISA, později rozšířeno i pro řadu IBM 7000
  - populárním programem na 1401 byl simulátor IBM 650, který umožnil zákazníkům spouštět mnoho programů pro IBM 650 na emulované IBM1401
    - (*650 simulovaná na 1401 emulovaná na 360)*

Mikroprogramování a 70-tá léta

- Byly vyvinuty velmi rychlé paměti ROM, rychlejší než dostupné DRAM
- Pro složitější instrukční soubory vycházely datové cesty *levnější a jednodušší*
- *Nové instrukce* , např., floating point, lze podporovat bez modifikace datových cest
- Bylo snazší *odhalovat a ladit* chyby v návrhu řadiče
- Kompatibilita ISA v rámci celé řady modelů je snáze dosažitelná a hlavně levněji dosažitelná

Vyjma nejlevnějších a nejrychlejších strojů, všechny počítače měly mikroprogramové řízení

Zapisovatelná řídící paměť

- Implementace řídící paměti pomocí RAM nikoliv ROM
  - paměti MOS SRAM jsou skoro tak rychlé jako řídící paměti (jádrové paměti nebo DRAM byly 2-10x pomalejší)
  - Je obtížné psát mikroprogramy bez chyb

- Uživatelsky zapisovatelné řídící paměti (WCS) byly volitelné u celé řady minipočítačů
  - Umožňovaly uživatelům změnu mikrokódu každého procesoru

- Uživatelské-WCS *selhaly*
  - Slabé a nebo dokonce žádné programovací prostředky
  - Obtížné vtěstnat software do malého prostoru
  - Mikroarchitektura byla přizpůsobena originální ISA, méně už uživatelské
  - Velké WCS jsou částí stavu procesoru – drahé přepínání kontextu
  - Obtížná ochrana v případě, že uživatel mění mikrokód
  - Virtuální paměť vyžadovala *restartovatelný* mikrokód

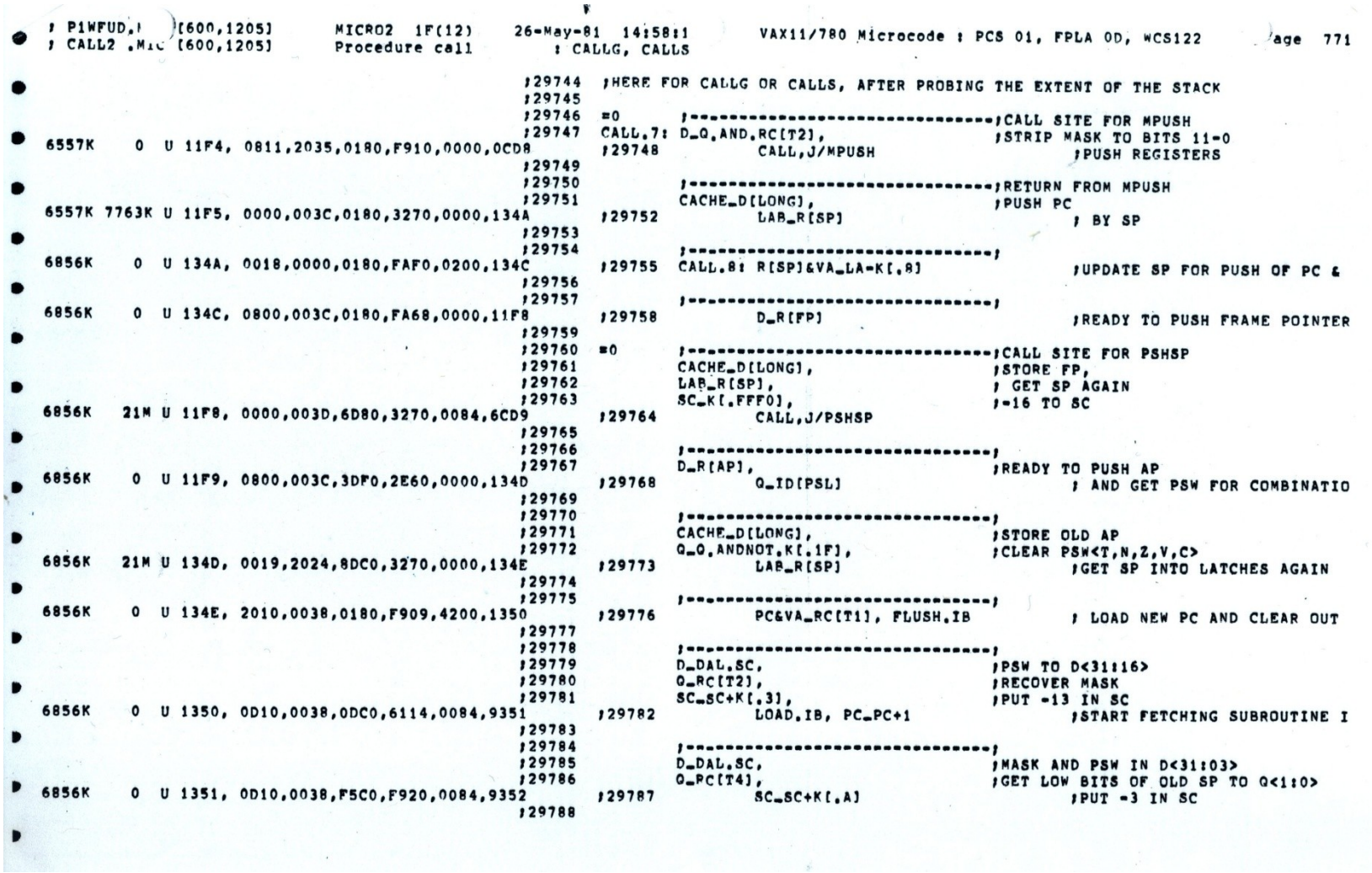
Mikroprogramování: počátek 80-tých let

- Evoluce přinesla mnohem složitější mikro-stroje
  - Složité instrukční soubory vynutily používání „podmikroprogramů“ a stacku na úrovni µkódu
  - Potřeba korigovat chyby v řídících programech byla v přímém rozporu s charakterem µROM
  - --> WCS (B1700, QMachine, Intel i432, …)
- S příchodem technologie VLSI předpoklady o rychlosti ROM & RAM přestaly být relevantní -> větší složitost
- Lepší kompilátory způsobily, že složité instrukce ztratily na důležitosti.
- Využití četných mikro-architekturálních inovací, např. pipelinningu, cache pamětí a bufferů způsobilo, že vícecyklové provádění instrukcí reg-reg se stalo neatraktivní
- RISC procesory v nejbližší budoucnosti
  - Využití plochy čipu pro rychlé instrukční cache pro uživatelsky-využitelné vertikální mikroinstrukce – využití softwarových rutin a nikoliv hardwarových mikrorutin
  - Využití jednoduchých ISA kvůli podpoře harwarových implementací s pipelinningem

Moderní využití

- *Mikroprogramování má daleko k „vyhynutí“*
- Hrálo významnou roli v počítačích 80-tých let
  DEC uVAX, řada Motorola 68K, Intel 386 a 486
- Mikroprogramování hraje stále asistenční roli u všech moderních mikroprocesorů *(AMD Athlon, Intel Core 2 Duo, IBM PowerPC)*
  - Většina instrukcí je prováděna s přímým HW řízením
  - Méně často využívané a složité instrukce aktivují mikroprogramový řadič
- *Modifikovatelný („patchable“)* mikrokód, typicky pro poslední fáze výroby, umožňuje korekci chyb
  např. Pentia fy Intel načítají „patches“ μkódu při bootu

Ukázka - mikrokód VAX 11-780



```
; P1WFUD.[ ][600,1205]      MICRO2  1F(12)    26-May-81  14:58:1      VAX11/780 Microcode : PCS 01, FPLA 0D, WCS122      Page  771
; CALL2 .MIC [600,1205]     Procedure call       : CALLG, CALLS

                                                ;29744  ;HERE FOR CALLG OR CALLS, AFTER PROBING THE EXTENT OF THE STACK
                                                ;29745
                                                ;29746  =0      ;-------------------------------;CALL SITE FOR MPUSH
                                                ;29747  CALL.7: D_Q,AND,RC[T2],                 ;STRIP MASK TO BITS 11-0
6557K    0  U 11F4, 0811,2035,0180,F910,0000,0CD8       ;29748          CALL,J/MPUSH                    ;PUSH REGISTERS
                                                ;29749
                                                ;29750          ;-------------------------------;RETURN FROM MPUSH
                                                ;29751          CACHE_D[LONG],                  ;PUSH PC
6557K 7763K U 11F5, 0000,003C,0180,3270,0000,134A       ;29752          LAB_R[SP]                       ; BY SP
                                                ;29753
                                                ;29754          ;-------------------------------;
6856K    0  U 134A, 0018,0000,0180,FAF0,0200,134C       ;29755  CALL.8: R[SP]&VA_LA-K[,8]               ;UPDATE SP FOR PUSH OF PC &
                                                ;29756
                                                ;29757          ;-------------------------------;
6856K    0  U 134C, 0800,003C,0180,FA68,0000,11F8       ;29758          D_R[FP]                         ;READY TO PUSH FRAME POINTER
                                                ;29759
                                                ;29760  =0      ;-------------------------------;CALL SITE FOR PSHSP
                                                ;29761          CACHE_D[LONG],                  ;STORE FP,
                                                ;29762          LAB_R[SP],                      ; GET SP AGAIN
                                                ;29763          SC_K[,FFF0],                    ;-16 TO SC
6856K   21M U 11F8, 0000,003D,6D80,3270,0084,6CD9       ;29764          CALL,J/PSHSP
                                                ;29765
                                                ;29766          ;-------------------------------;
                                                ;29767          D_R[AP],                        ;READY TO PUSH AP
6856K    0  U 11F9, 0800,003C,3DF0,2E60,0000,134D       ;29768          Q_ID[PSL]                       ; AND GET PSW FOR COMBINATIO
                                                ;29769
                                                ;29770          ;-------------------------------;
                                                ;29771          CACHE_D[LONG],                  ;STORE OLD AP
                                                ;29772          Q_Q,ANDNOT,K[,1F],              ;CLEAR PSW<T,N,Z,V,C>
6856K   21M U 134D, 0019,2024,8DC0,3270,0000,134E       ;29773          LAB_R[SP]                       ;GET SP INTO LATCHES AGAIN
                                                ;29774
                                                ;29775          ;-------------------------------;
6856K    0  U 134E, 2010,0038,0180,F909,4200,1350       ;29776          PC&VA_RC[T1], FLUSH,IB          ; LOAD NEW PC AND CLEAR OUT
                                                ;29777
                                                ;29778          ;-------------------------------;
                                                ;29779          D_DAL,SC,                       ;PSW TO D<31:16>
                                                ;29780          Q_RC[T2],                       ;RECOVER MASK
                                                ;29781          SC_SC+K[,3],                    ;PUT -13 IN SC
6856K    0  U 1350, 0D10,0038,0DC0,6114,0084,9351       ;29782          LOAD,IB, PC_PC+1                ;START FETCHING SUBROUTINE I
                                                ;29783
                                                ;29784          ;-------------------------------;
                                                ;29785          D_DAL,SC,                       ;MASK AND PSW IN D<31:03>
                                                ;29786          Q_PC[T4],                       ;GET LOW BITS OF OLD SP TO Q<1:0>
6856K    0  U 1351, 0D10,0038,F5C0,F920,0084,9352       ;29787          SC_SC+K[,A]                     ;PUT -3 IN SC
                                                ;29788
```

Odkazy

- Pro přípravu lekce byly použity kromě dalších hlavně materiály:
  - Tannenbaum: Structured computer organization
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)