Úvod do organizace počítače

Řízení multicyklové jednotky

Mikroprogramování a výjimky (přerušení)

Opakování – Multicyklová jednotka

- **Každá instrukce se provádí ve více stupních**
- **Zpracování v jednom stupni trvá jeden cykl**
  1. Načtení instrukce
  2. Dekódování instrukce / načtení dat
  3. Operace ALU provedení R-formátu
  4. Dokončení instrukce typu R
  5. Dokončení přístupu do paměti

Pro všechny instrukce společné

☺ **Každý stupeň může opět použít hardware předchozího stupně**

☺ *Efektivnější využití hardware a času*

>> Nový hardware + multiplexery pro oddělení stupňů a přepínání datových cest, řízení

>> Navíc řízení nového hardware

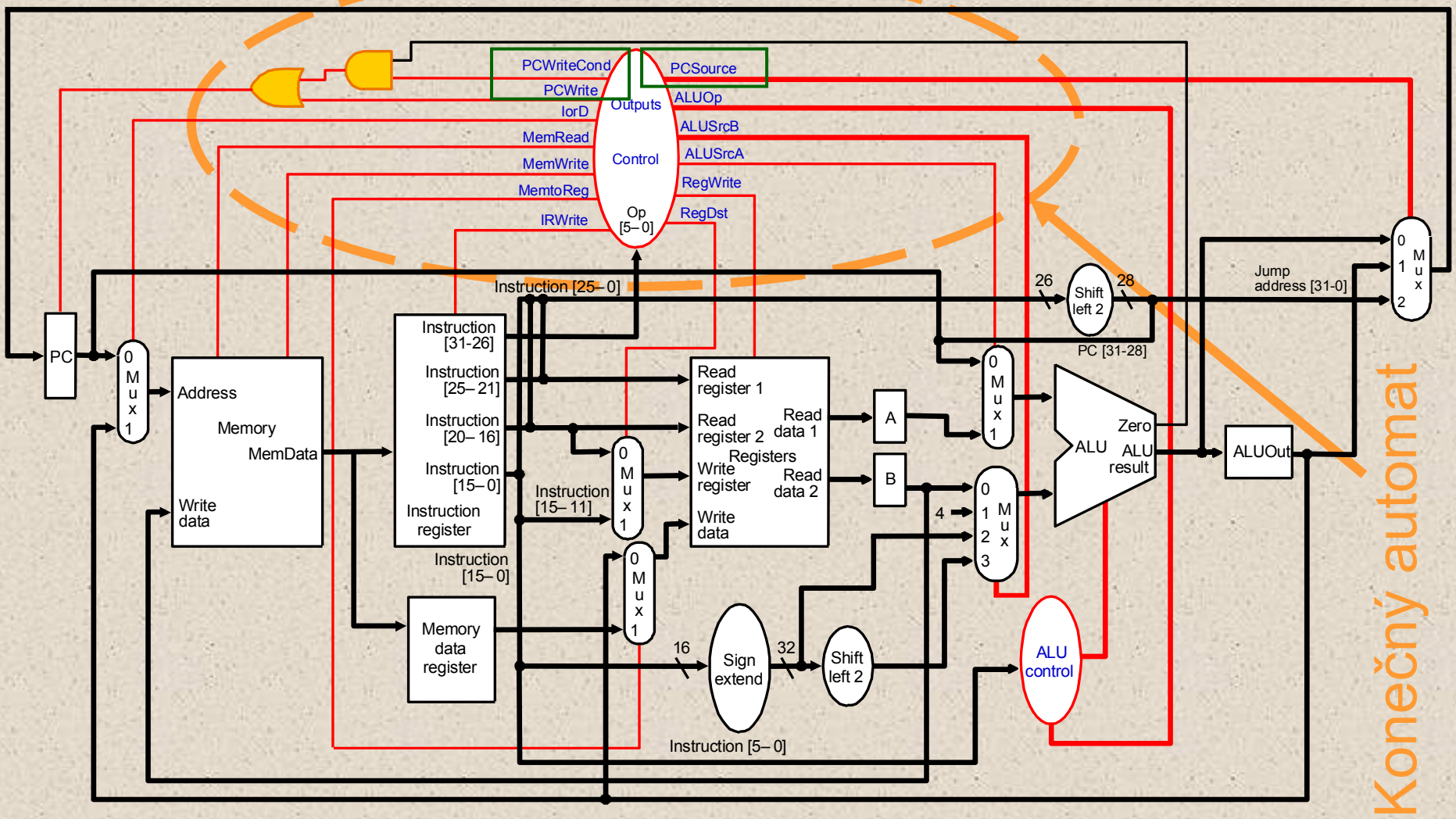
Opakování – Multicyklová jednotka

- Multicyklová jednotka – 1 cykl/krok :
  1. Načtení instrukce (Instruction Fetch)
  2. Dekódování instrukce / čtení dat
  3. Operace ALU / Provádění instrukcí formátu-R
  4. Dokončení instrukcí formátu-R
  5. Dokončení přístupu do paměti
- Konečný automat (Finite-State Machine)
  - současný stav, příští stav, přechodová funkce
- Řadič (konečný automat)
  - **Stav:** Generování řídících signálů
  - **Hrany:** Podmínky přechodu
  - Lze implementovat v hardware (ROM, PLA)

Přehled

- Problémy spojené s řízením
  - Reálné procesory mají stovky stavů
  - Tisíce interakcí mezi stavy
  - Grafický návrh automatu je pro reálné architektury nemožný
- Řešení: Mikroprogramování (Wilkesův automat)
  - Návrh založený na „softwarových principech“ (firmware)
  - Řídící signály generované při vykonávání ***mikroinstrukce*** => určeny výstupním polem *mikroinstrukce*
  - Posloupnost mikroinstrukcí => ***mikroprogram***
- Mikroprogramování
  - Návrh mikroinstrukcí a jejich časování
  - Ošetření výjimek

Řízení multicyklové jednotky



Multicyklové jednotky: 1-bitové řídící signály

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None	The output of the memory is written into the IR.
PCWrite	None	The PC is written; the source is controlled by PCSource.
PCWriteCond	None	The PC is written if the Zero output from the ALU is also active.

Multicyklové jednotky: 2-bitové řídící signály

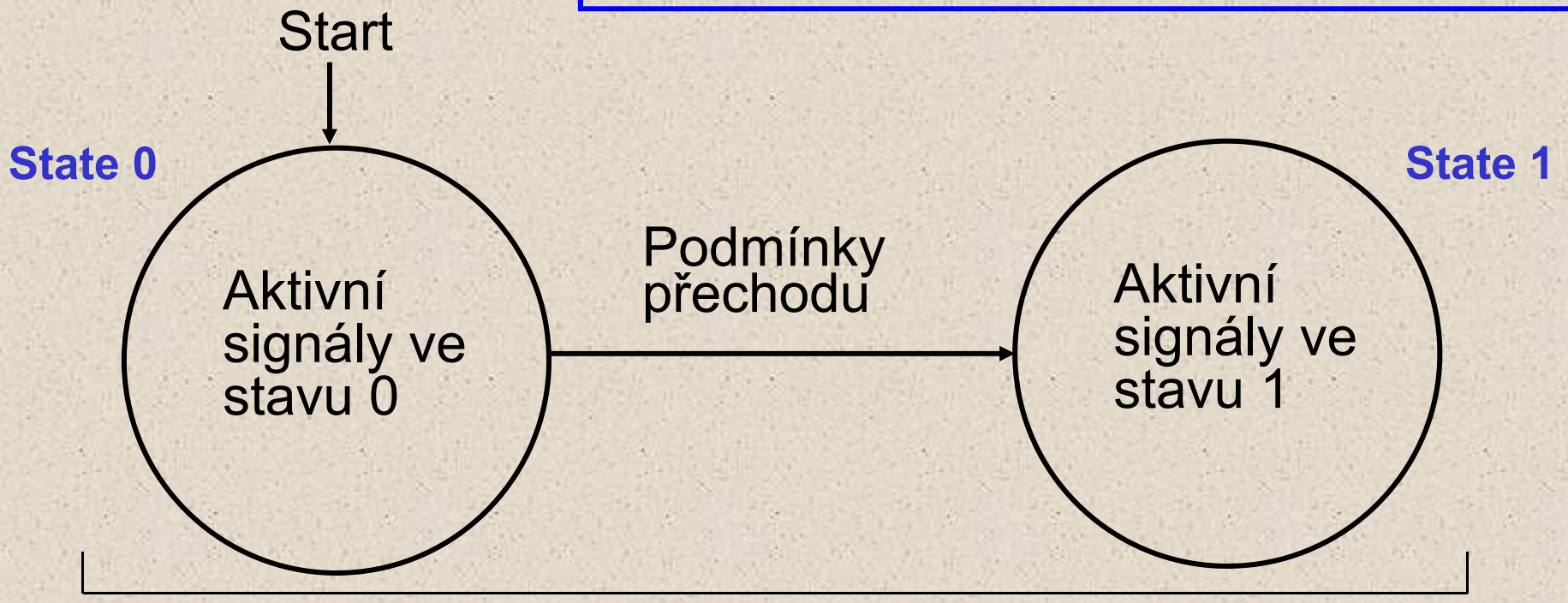
Signal name	Value	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25–0] shifted left 2 bits and concatenated with PC + 4[31–28]) is sent to the PC for writing.

Základy řízení pomocí FSM

Stav: „Snímek stroje“ (stav paměťových prvků)

Přechod: změna stavu (přechod od jednoho stavu do druhého

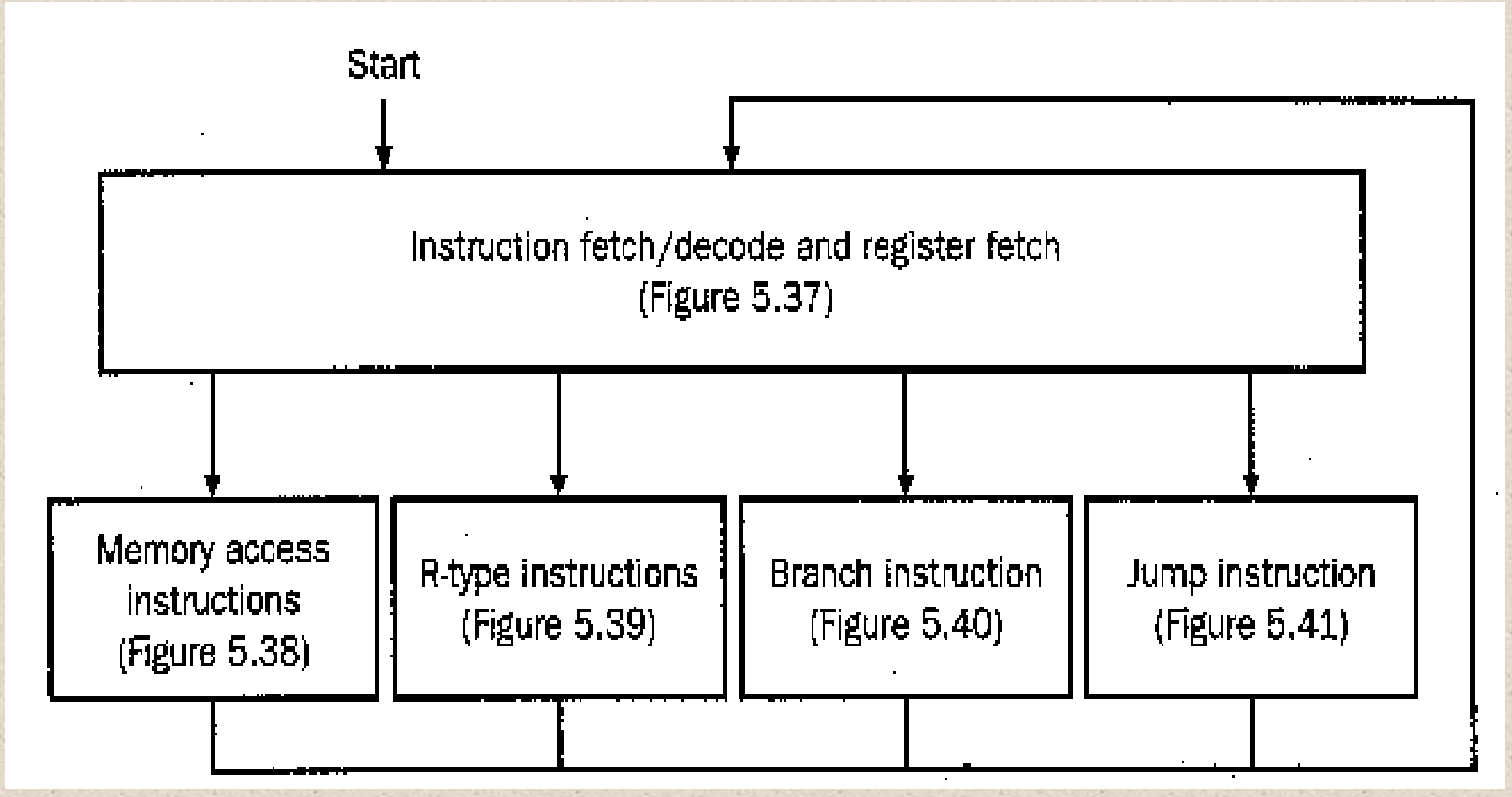
viz kánonický tvar konečného automatu



Finite State Machine – automat s konečným počtem stavů

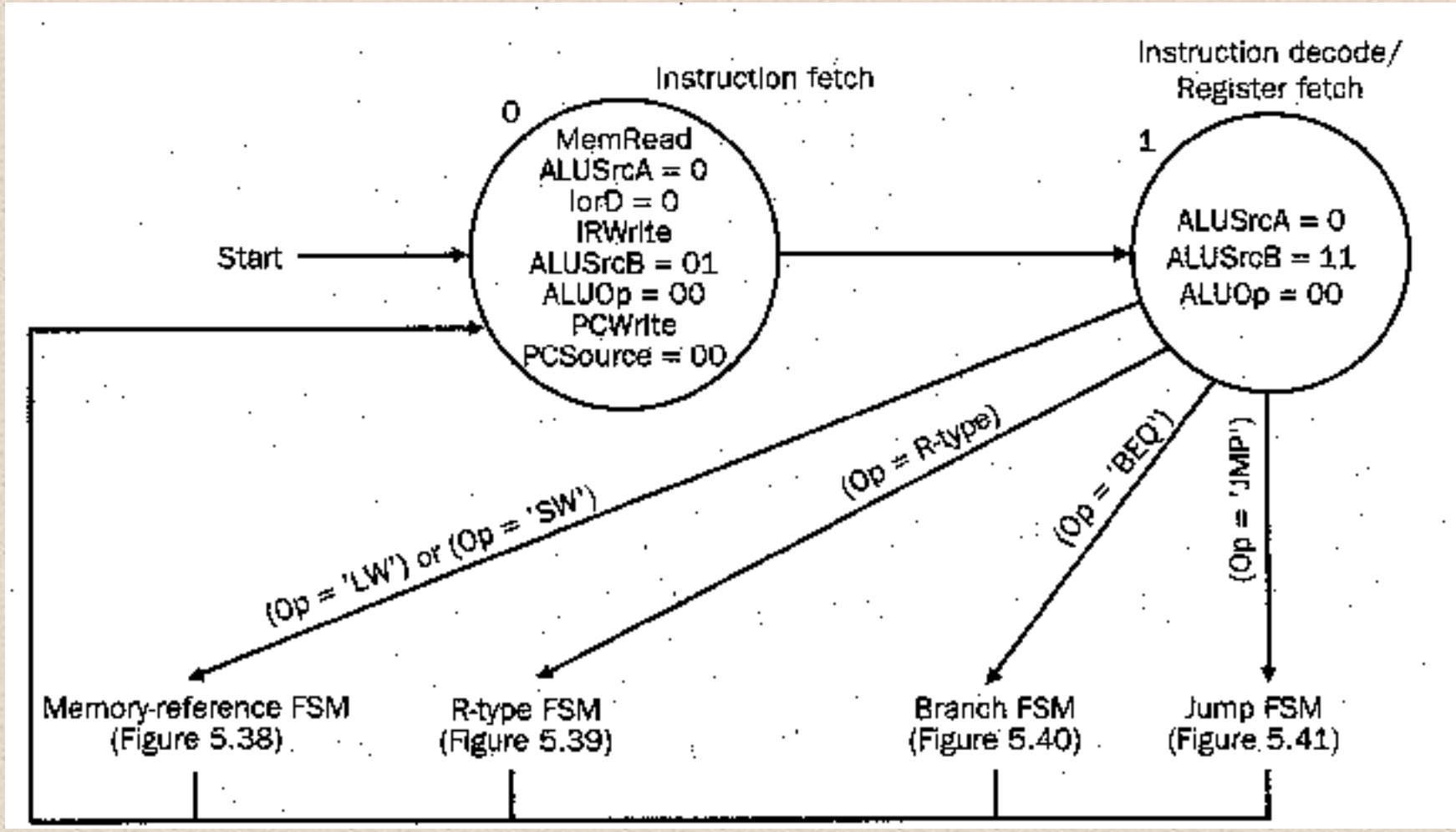
FSC pro multicyklové jednotky

„Pohled shora“ – vhodné pro abstrakci



FSC pro načtení & dekódování instrukce

Společné: Načtení instrukce/dat, dekódování



Multicyklová jednotka: R-formát

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole opcode, rd, rs, rt, funct

Načtení dat: Výběr registrů podle rs, rt

Data načtena do pomocných registrů A, B (vstup ALU)

Krok 3: Operace ALU (ALUsrcA, ALUsrcB, ALUop)

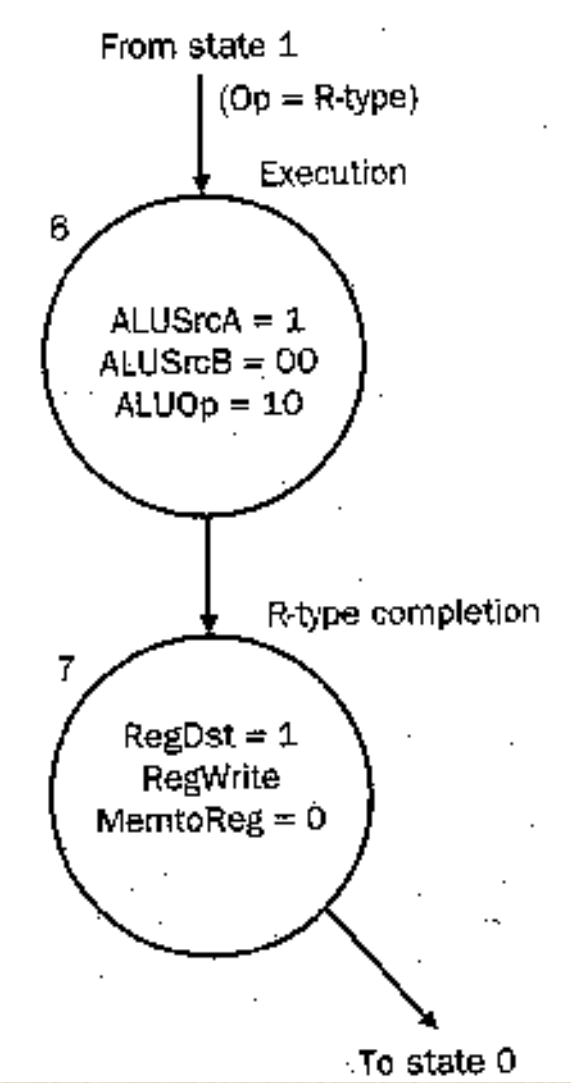
Výstup z ALU je zapsán do registru ALUout

Krok 4: Obsah registru ALUout jde na zápisový port registrové sady
V rd je zapsáno číslo registru

Aktivovány signály: RegWrite, RegDst

CPI pro R-formát = 4 cykly

FSC pro R-formát instrukcí



Předpokládáme ukončení Fetch/Decode fáze

Určeny vstupy ALU z bufferů A, B

Provedení operace ALU podle opcode

Určení cílového registru podle rd

Zápis bufferu ALUout do sady registrů

Zpět na zpracování další instrukce

Multicyklová jednotka: Store Word (sw)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole opcode, rs, rt, offset

Načtení dat: rt adresuje registrovou sadu => Bázová adresa

Data načtena do buffer registru A (báze)

SignExt, posun pole offset do buffer registru B

Krok 3: Operace ALU (ALUsrcB, ALUop) => Báze + Offset

výstup ALU jde do registru ALUout

Krok 4: Obsah registru ALUout je použit jako adresa do paměti

Aktivován signál: MemWrite [ALUout => reg. sadu]

CPI pro Store = 4 cykly

Multicyklová jednotka: Load Word (lw)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole opcode, rd, rt, offset

Načtení dat: rt – pointer do registrové sady => Bázová adresa

Data načtena do buffer registru A (báze)

SignExt, posun pole offset do buffer registru B

Krok 3: Operace ALU (ALUsrcB, ALUop) => Báze + Offset

výstup ALU jde do registru ALUout

Krok 4: Obsah registru ALUout je použit jako adresa do paměti

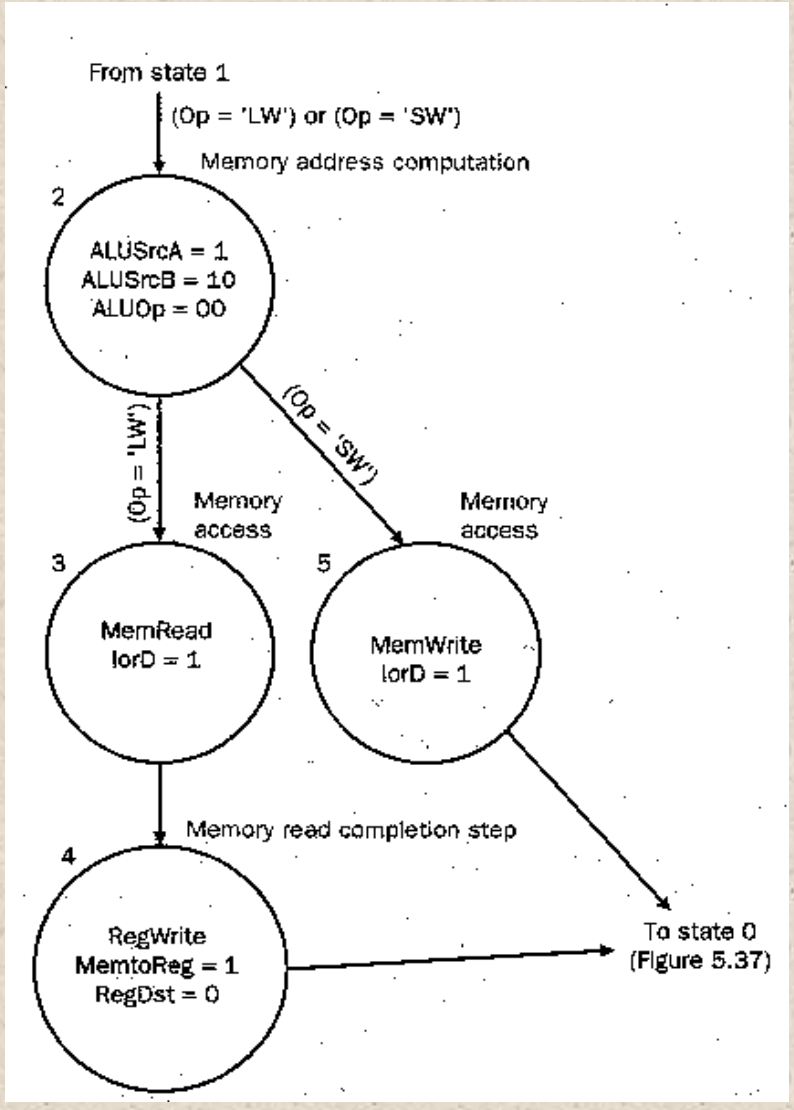
Aktivován signál: MemRead

Krok 5: Data z paměti jdou na zápisový port registrové sady,

adresa určena obsahem rd - proveden zápis

CPI pro Load = 5 cyklů

FSC pro instrukce Load/Store



Předpokládáme ukončení fáze Fetch/Decode

Určeny vstupy ALU z bufferů A, B

Provedení operace ALU - výpočet adresy do paměti (MemAddr)

Provedení přístupu do paměti (read/write)

If Load, then do Register Write

Zpět na zpracování další instrukce

Multicyklová jednotka: Podmíněný skok

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: pole opcode, rs, rt, offset

Načtení dat: rs a rt určují adresy do sady registrů

Výpočet BTA: SignExt, posun pole offset do buffer registru B

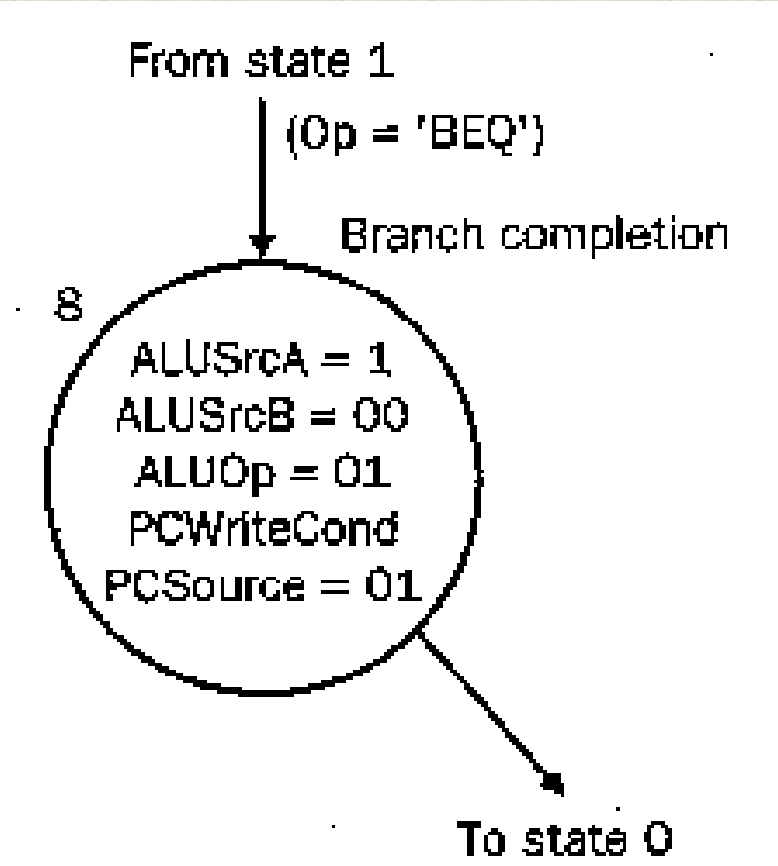
ALU určí PC, offset => BTA

Krok 3: Operace **ALU** (ALUsrcA, ALUsrcB, ALUop) = komparace podle výstupu z ALU (výsledek Zero) dojde k výběru BTA nebo PC+4

CPI pro podmíněný skok = 3 cykly

Pozn.: BTA … adresa cíle instrukce větvení

FSC pro podmíněný skok



Předpokládáme ukončení fáze Fetch/Decode

Určeny vstupy ALU z bufferů A, B

Provedení operace ALU => BTA

Zápis BTA nebo PC+4 do PC

Zpět na zpracování další instrukce

Multicyklová jednotka: skok (Jump)

Krok 1: Načtení instrukce // Uložena do IR // Výpočet PC + 4

Krok 2: Dekódování instrukce: Pole opcode, address

Výpočet JTA: Pole SignExt, Shift offset [Bity 27-0]

sestaveny z části PC [Bity 31-28] => JTA

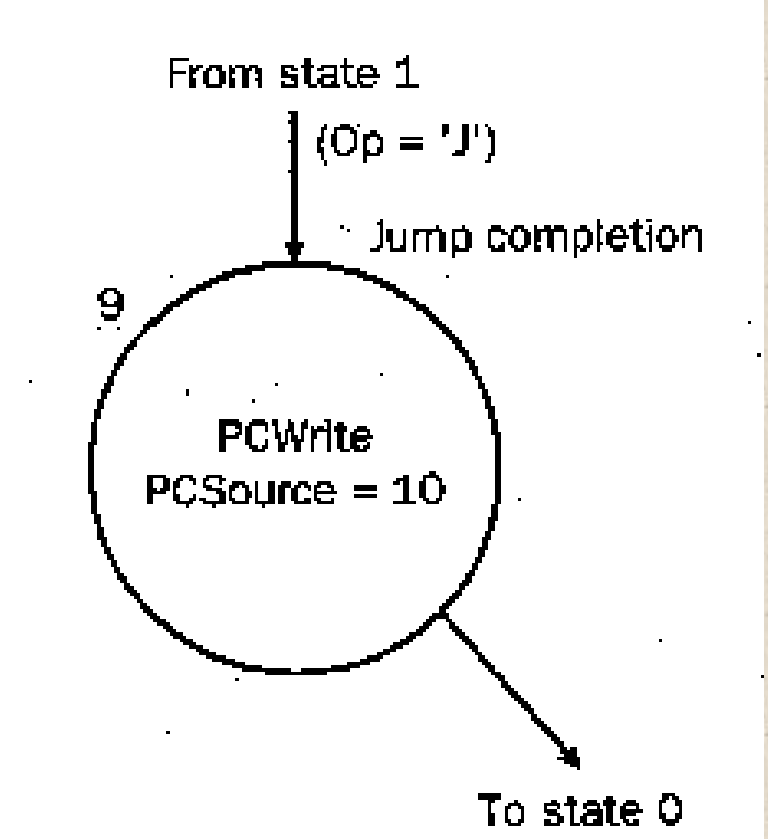
Krok 3: Obsah PC nahrazen cílovou adresou skoku (JTA)

PCsource = 10, *Aktivován signál:* PCWrite

CPI pro Jump = 3 cykly

Pozn.: JTA … adresa cíle instrukce skoku

FSC pro instrukci skoku



Předpokládáme ukončení fáze Fetch/Decode

PC bude přepsán

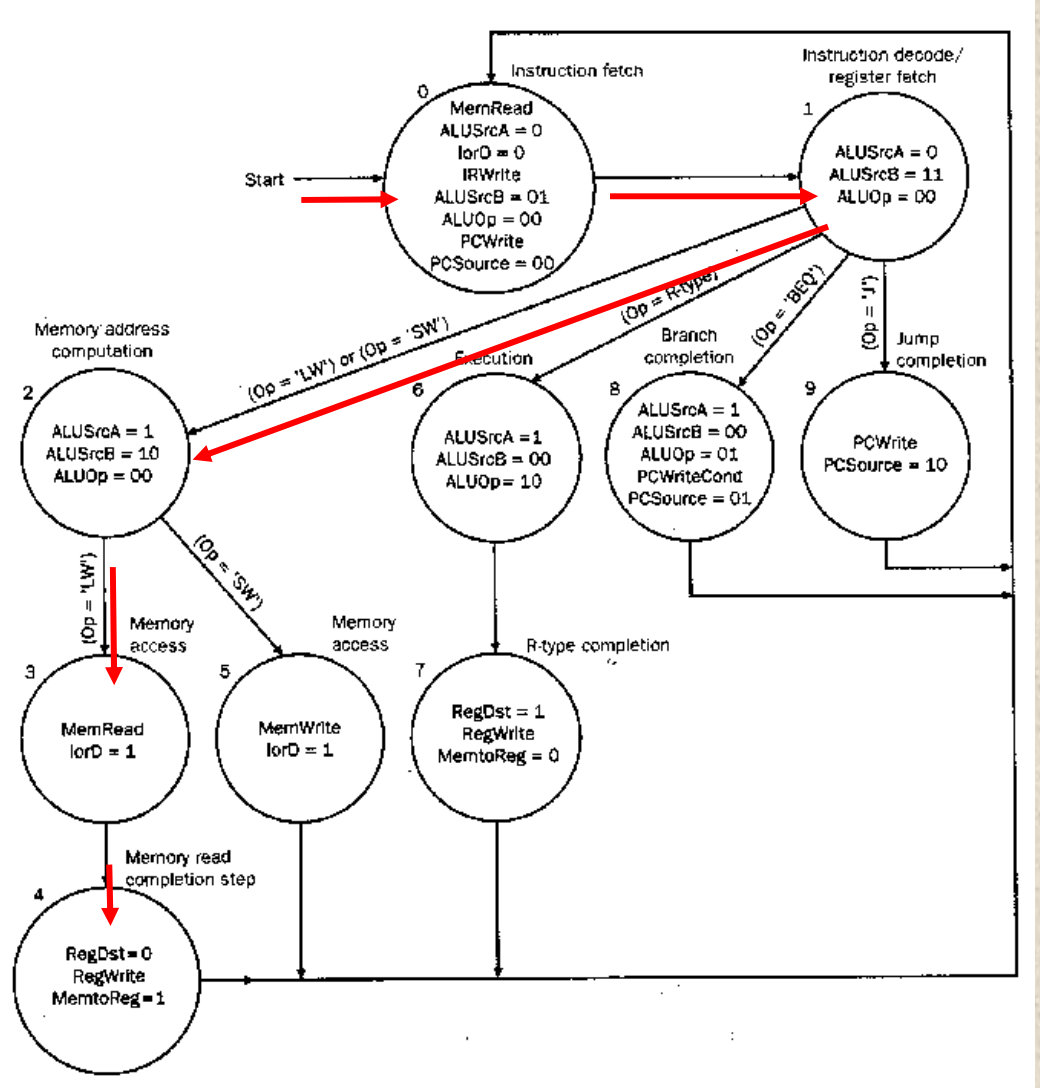
Hardware sestaví JTA

JTA se zapíše do PC

Zpět na zpracování další instrukce

JTA … Jump Target Address

Kompletní FSC



10 stavů celkem

CPI = počet stavů nutných pro provedení dané instrukce

R-formát = 4 stavy

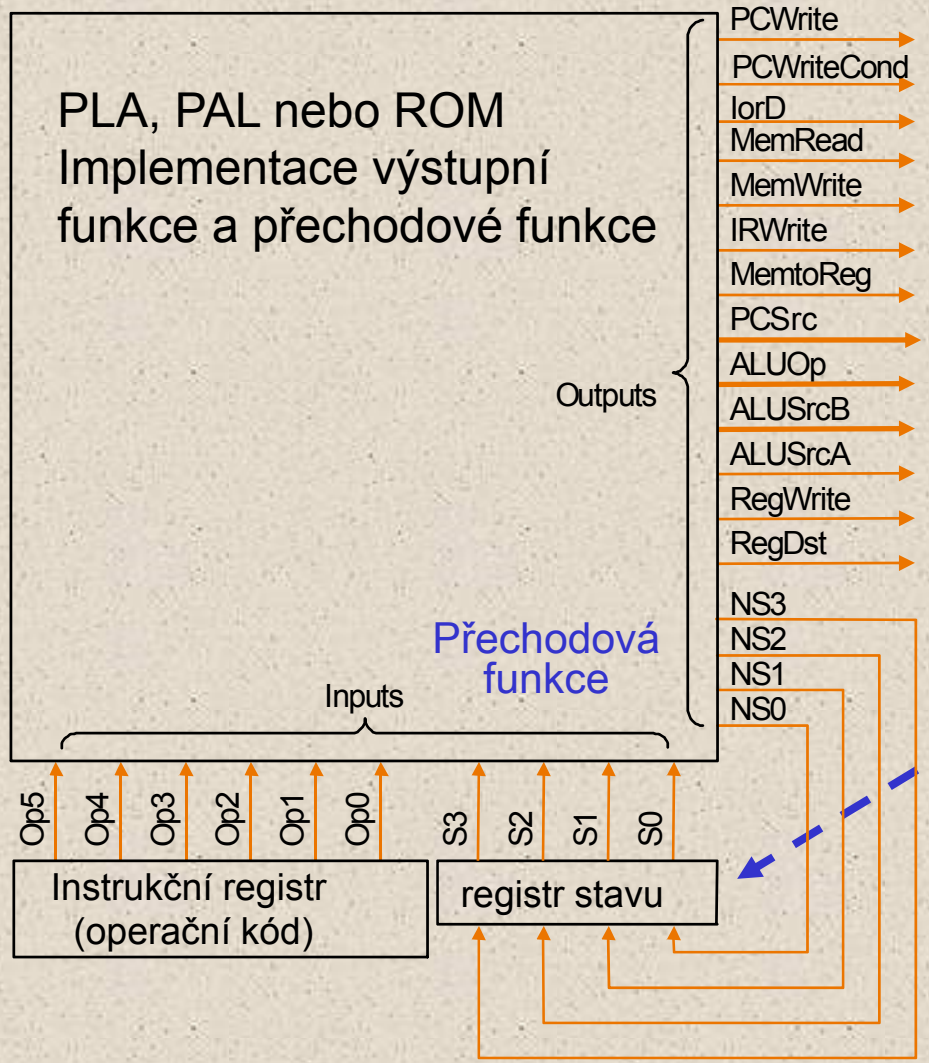
Store = 4 stavy

Load = 5 stavů [0,1,2,3,4]

Branch = 3 stavy

Jump = 3 stavy

Hardware pro multicyklovou FSC

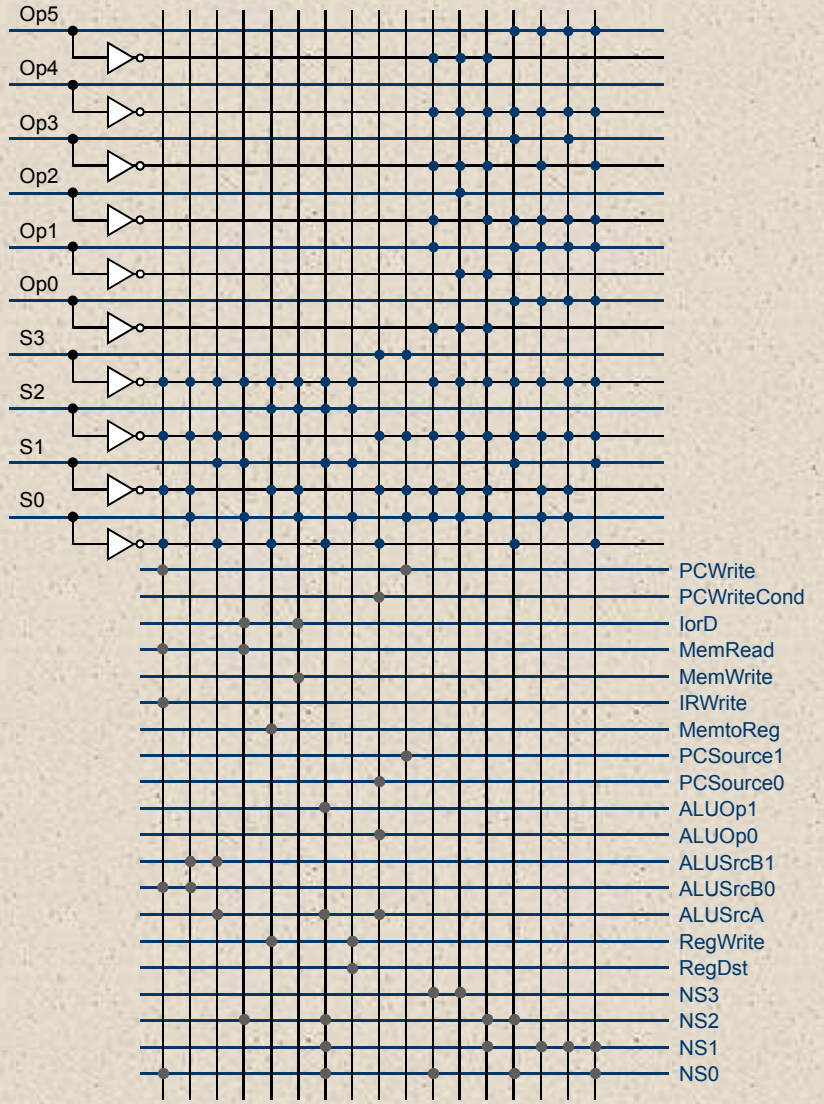


Programmable Array Logic (PAL)

Programovatelné logické pole (PAL)

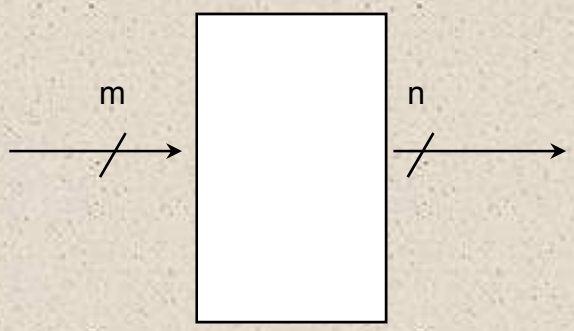
Registr, kde je zapsán stav konečného automatu

Implementace pomocí PLA



Implementace pomocí ROM

- ROM = "Read Only Memory"
  - Hodnoty jsou zapsány a nelze je měnit
- Pamět ROM lze využít k implementaci pravdivostní tabulky
  - Má-li adresa m-bitů, můžeme adresovat 2^m položek v ROM.
  - Výstupem jsou data, na které ukazuje adresa.
    m je „výška" a n je „šířka“ , odpovídající počtu výstupů.



Implementace pomocí ROM

- Kolik je v našem příkladu vstupů?
  6 bitů operační kód, 4 bity pro stav => 10 adresních linek
  (t.j. 2^{10} = 1024 různých adres)
- Kolik je výstupů?
  16 výstupních řídících signálů, 4 stavové bity => 20 výstupů

- Organizace ROM je 2^{10} x 20 = 20K bitů (trochu neobvyklá velikost a proto bereme nejbližší vyšší)

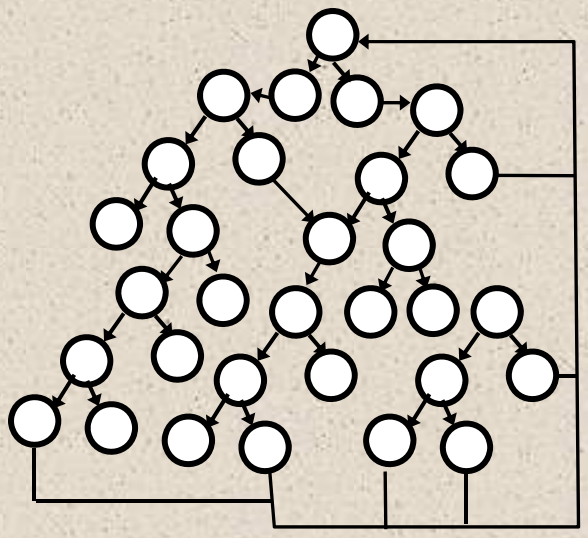
- Velmi nehospodárné vzhledem k velkému počtu situací, které nás nezajímají => výstupy závisejí pouze na stavech, nikoliv na op. kódu.

ROM versus PLA

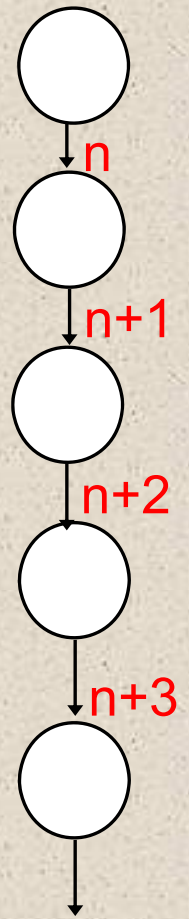
- Rozdělíme tabulku na dvě části
  — 4 stavové bity určují celkem 16 výstupů, 2^4 x 16 bitů ROM
  — 10 bitů určuje 4 bity příštího stavu, 2^{10} x 4 bitů ROM
  — Celkem: 4.3K bitů ROM => poměrně značná úspora.
- PLA je mnohem menší
  — může sdílet součinové termy
  — obsahuje jen položky, které produkují aktivní výstup
  — ošetřuje i případy (don't cares)
- Velikost je rovna (#vstupů x #produktových-termů) + (#výstupů x #produktových-termů)
  V tomto případě = (10x17)+(20x17) = 510 PLA buněk,
  buňka PLA je téměř tak velká jako buňka ROM (trochu větší).

Alternativa k FSM pro multicyklovou verzi?

- MIPS-lite má (asi) 7 instrukcí, 10 FSM stavů
- Reálné stroje mají 100 a více instrukcí; reálné řadiče mají stovky až tisíce stavů!
- Problém: FSM bublinový diagram by byl příliš velký



Pozorování na reálných strojích



- Strojový jazyk: příští instrukce je určena implicitně.
  - PC registr určuje instrukci
  - Příští instrukce leží vždy na adrese PC+4 (vyjma skoku)
- FSM řadič: často produkuje jen jeden přechod od současného k příštímu stavu
- Vypůjčíme-li si tuto myšlenku z instrukční úrovně, bude každý řídící krok znamenat určitý druh “instrukce”?
- To vede na mikroprogramové řízení

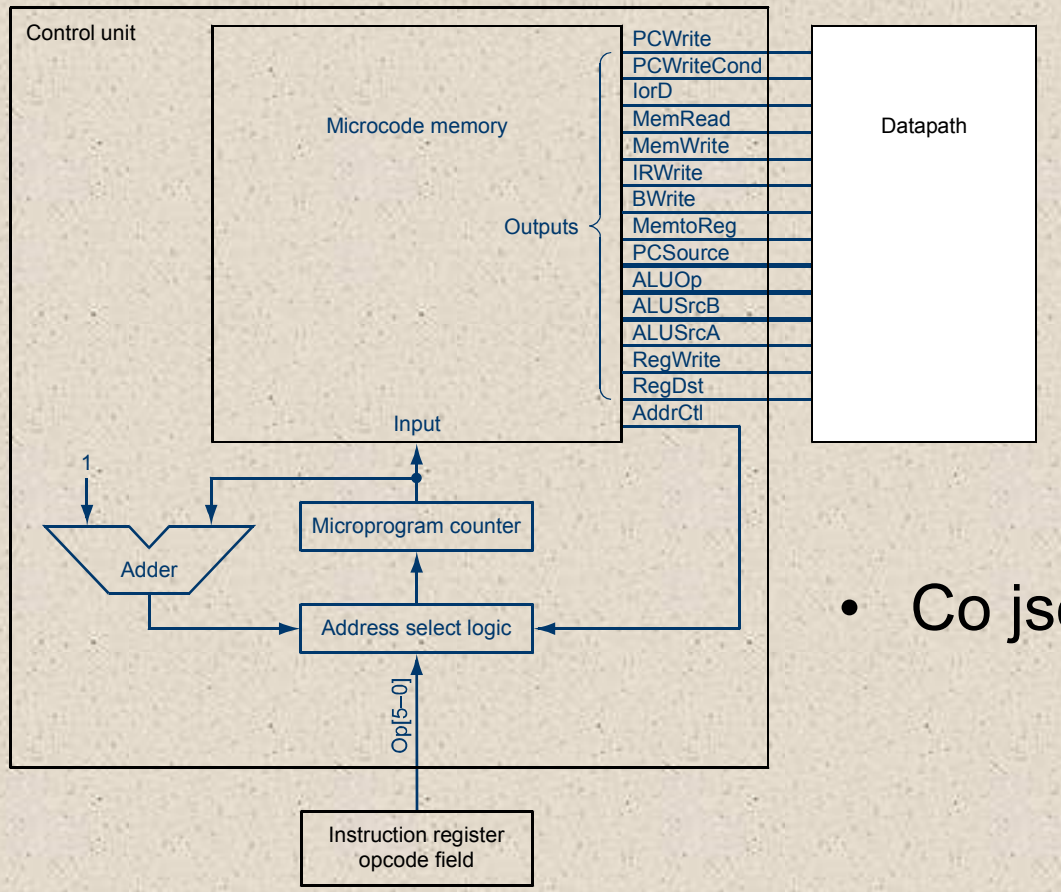
Mikroprogramové řízení

- U mikroprogramového řízení vlastně stavy FSM přechází na mikroinstrukce mikroprogramu (“mikrokód”)
  - Jeden stav FSM = jedna mikroinstrukce
  - Pomocí mikroinstrukcí jsou interpretovány instrukce
- Stavový registr FSM začne hrát roli čítače mikroinstrukcí (mPC)
  - Normální provádění: přičti 1 k mPC => adresa další mikroinstrukce
  - Větvení mikroprogramu: další logika určuje příští mikroinstrukci

Mikroprogramování vs HW řízení

- Mikroprogramování přináší flexibilitu pro návrh a změny architektury. Řídící paměť (ROM) lze přeprogramovat nebo nahradit. Hardwarové řízení se obtížně navrhuje, je-li soubor instrukcí složitý. Po dokončení návrhu nejsou změny možné.
- Mikroprogramování je pomalejší, protože se do řídící paměti přistupuje v každém cyklu. Přístup do paměti je pomalý. Hardwarové řízení je rychlé, protože doba cyklu závisí pouze na zpoždění kombinačních obvodů řídící jednotky, které je mnohem kratší než doba přístupu do paměti.

Mikroprogramování



- Co jsou “mikroinstrukce”?

Mikroprogramování

- Specifikační metodologie
  - Vhodné pro stovky operačních kódů a množství adresních režimů
  - Signály se specifikují symbolicky pomocí mikroinstrukcí

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

Mají dvě implementace stejné architektury stejný mikrokód (firmware)?
Jakou úlohu plní mikroasembler?

Mikroinstrukční formát

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Maximální vs. minimální kódování

- Žádné kódování:
  - 1 bit pro každý signál mikrooperace
  - rychlé, vyžaduje více paměti (logika)
  - použito pro Vax 780 — celých 400K paměti!
- Výrazné kódování:
  - Signály mikrooperací se získávají dekódováním výstupního pole
  - Méně paměti, ale pomalejší, menší míra paralelizmu
- Historický kontext procesorů CISC:
  - Příliš mnoho logiky, která by se měla umístit na jeden chip spolu s ostatním
  - Použití ROM (nebo i RAM) pro uložení mikrokódu
  - Je jednoduché přidávat další instrukce

Mikrokód: Kompromisy

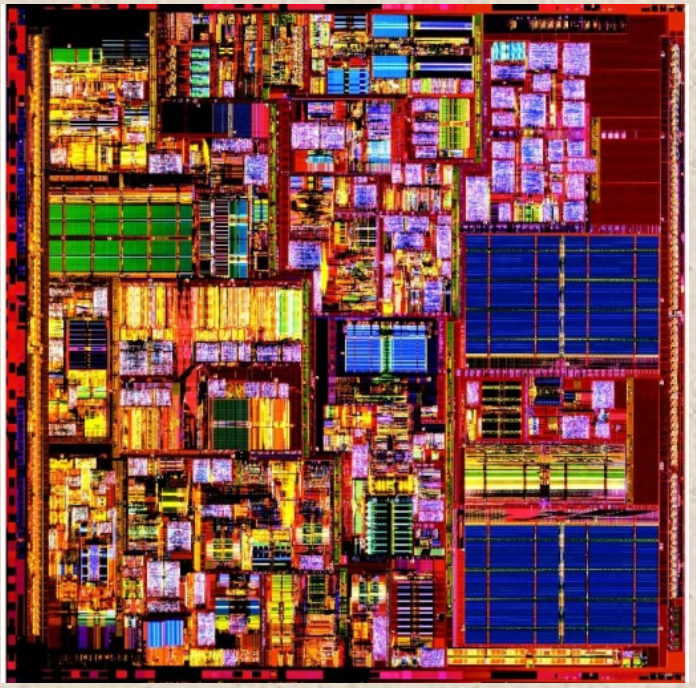
- Rozdíl mezi specifikací a implementací je někdy „neostrý“
- Výhody specifikace:
  - Snadný návrh a psaní
  - Současný návrh architektury a mikrokódu
- Výhody implementace (v externí ROM)
  - Snadná změna, protože se jedná o změnu obsahu paměti
  - Lze emulovat jiné architektury
  - Lze využívat interní registry
- Nevýhody implementace, dnes POMALÉ, protože:
  - Řadič bývá implementován na stejném čipu jako zbytek procesoru
  - ROM není dnes rychlejší než RAM
  - Obvykle není třeba se vracet a dělat změny

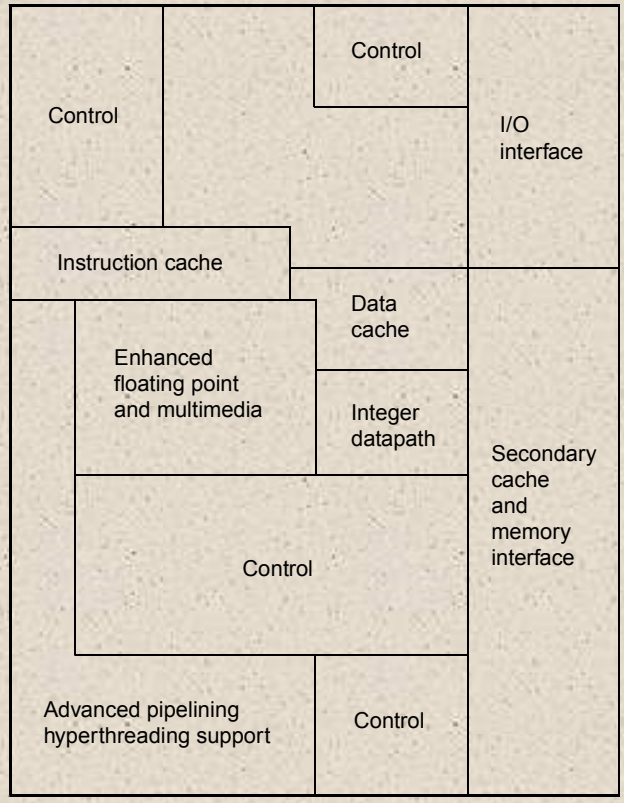
Historický přehled

- V 60-tých a 70-tých letech bylo mikroprogramování velmi důležité při implementaci počítačů
- Vedlo na velmi propracované ISA, např. na VAX
- V 80-tých letech se staly populární procesory RISC, založené na hlubokém pipeliningu
- Pipelining je u mikroprogramování také možný!
- Implementace architektury procesoru IA-32 počínaje 486 používala:
  - "hardwarové řízení" jednoduchých instrukcí
    (malý počet cyklů, FSM implementován využitím PLA nebo „random logic")
  - "mikroprogramové řízení" složitějších instrukcí (velký počet cyklů, centrální řídící paměť)
- Architektura IA-64 využívá styl RISC ISA a může být implementována bez velké centrální řídící paměti.

Random logic is a semiconductor circuit design technique that translates high-level logic descriptions directly into hardware features such as AND and OR gates. The name derives from the fact that few easily discernible patterns are evident in the arrangement of features on the chip and in the interconnects between them. In VLSI chips, random logic is often implemented with standard cells and gate arrays.

Pentium 4





- Někde musí být ošetřeny i složité instrukce.
- Procesor provádí jednoduché mikroinstrukce, 70 bitů široké (hardwired).
- 120 řídících linek pro integer jednotku (400 pro floating point).
- Jestliže některá instrukce vyžaduje pro implementaci více než 4 mikroinstrukce, řízení pak přichází z řídící paměti ROM (8000 mikroinstrukcí). **Složité!**

Přehled

- **MIPS ISA**: Tři instrukční formáty (R, I, J)
- Jeden cykl/stupeň, každý formát - různé stupně
- **Jednocyklové kroky (akce)**

	R-fmt	lw	sw	beq	j
1. Načtení instrukce	■	■	■	■	■
2. Dekódování instrukce / čtení dat	■	■	■	■	■
3. Operace ALU/provedení R-formátu	■	■	■	■	■
4. Dokončení R-formátu	■	■	■		
5. Dokončení přístupu do paměti		■			

Konečný automat zjednodušuje návrh řízení

Formát mikroinstrukcí MIPS

Mikroinstrukce: Abstrakce řízení datových cest

Pole	Specifikuje
• *ALU control*	operaci ALU v daném cyklu hodin
• *SRC1*	zdroj 1. operandu ALU
• *SRC2*	zdroj 2. operandu ALU
• *Register Control*	registr read/write, zapisovaná data
• *Memory*	read/write pro paměť, zapisovaná data
	cílový registr pro MemRead
• *PCWrite control*	zdroj pro PC (PC+4, BTA, JTA)
• *Sequencing*	výběr příští mikroinstrukce

Režimy výběru (sequencing)

Specifikuje výběr příští mikroinstrukce

- *Inkrementace* – Je-li aktuální adresa mikroinstrukce A, potom příští 32-bitová mikroinstrukce leží na A + 4 (hodnota = *Seq*)
- *Větvení* – Skok na mikroinstrukci, která načítá příští instrukci MIPS (hodnota = *Fetch*)
- *Řízený výběr* – Příští mikroinstrukce je vybrána podle vstupu řadiče (hodnota = *Dispatch i*)

Výběrové tabulky:

- Podobné *tabulce skoků* při programování MIPS
- Určuje mikroprogramovému řadiči, kde se startuje specifická rutina v mikroprogramu (např., přípravná fáze instrukce)

Načtení instrukce a dekódování

Mikroinstrukce:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
Fetch	Add	PC	4	---	Read PC	ALU	Seq
---	Add	PC	Extshft	Read	---	---	Dispatch 1

Akce:

1. ALU provede PC+4, přesune ALUout do PC, přečte příští mikroinstrukci
2. ALU sečte PC a znaménkem rozšířený posunutý offset, čímž určí BTA, potom čte data z registrové sady do bufferů A a B

Přístup do paměti

Mikroinstrukce:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
Mem1	Add	A	Extend	---	---	---	Dispatch 2
LW2	---	---	---	---	Read ALU	---	Seq
---	---	---	---	Write MDR	---	---	Fetch

Akce:

1. ALU sečte bázi v A + rozšířený offset => adresa do paměti
2. Při čtení se paměť čte na adrese = ALUout
3. Výstup z paměti se zapíše do datového registru paměti (MDR)

   (Instrukce zápisu je symetrická)

Provedení instrukce R-formátu

Mikroinstrukce:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
Rformat1	Func code	A	B	---	---	---	Seq
---	---	---	---	Write ALU	---	---	Fetch

Akce:

1. Operace ALU je specifikována polem v mikroinstrukci *funct,* pracuje s A a B – výsledek je zapsán do registru ALUout
2. Výsledek v ALUout je zapsán do registrové sady a provede se skok zpět k místu volání tak, aby se načetla další instrukce MIPS

Větvení a skoky

Mikroinstrukce:

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
Beq1	Subt	A	B	---	---	ALUout-cond	Fetch
Jump1	---	---	---	---	---	Jump address	Fetch

Akce:

- *Branch* provede A-B v ALU tak, aby se nastavil výstup Zero v případě, že A=B, potom se skáče na BTA, vypočítanou během kroku dekódování instrukce
- *Jump* – skok se provede zápisem JTA do PC
- Obě mikroinstrukce se navrací do bodu, odkud byly volány pomocí výběrové tabulky (např., volání Beq1 nebo Jump1)

Kompletní mikroprogram

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite	Sequencing
Fetch	Add	PC	4	---	Read PC	ALU	Seq
---	Add	PC	Extshft	Read	---	---	Dispatch 1
Mem1	Add	A	Extend	---	---	---	Dispatch 2
LW2	---	---	---	---	Read ALU	---	Seq
---	---	---	---	Write MDR	---	---	Fetch
SW2	---	---	---	---	Write ALU	---	Fetch
Rformat1	Func code	A	B	---	---	---	Seq
---	---	---	---	Write ALU	---	---	Fetch
Beq1	Subt	A	B	---	---	ALUout-cond	Fetch
Jump1	---	---	---	---	---	Jump address	Fetch

- **Výběrová tabulka 1:** Mem1, Rformat1, Beq1, Jump1
- **Výběrová tabulka 2:** LW2 (load), SW2 (store)

Problémy mikroprogramování

- Hardwarová implementace
  - Podobné řízení s pevným řadičem (FSC)
  - Stav uložen v registru, přechodová funkce v ROM nebo PLA
  - *Možnosti*: mprog sequencer používá *čítač (counter)*
- Nesprávná představa: Mikroprogram je rychlejší
  - Kdysi: Paměť mikroprogramu tvořila rychlá paměť
  - Dnes: Použití cache, výhoda se ztrácí
  - Základ: Je snazší měnit software než hardware
- Omyl: Nové mikroinstrukce jsou “zadarmo”
  - Paměť mikroprogramu nemusí být zcela zaplněna (pro začátek)
  - Později lze přidávat další instrukce

Nové: Výjimky a interrupty

Definice:

Událost, způsobující změnu toku instrukcí mimo normální běh programu.

Typy: (1) Výjimka [overflow]

(2) Interrupt [I/O]

Rozdíly: *Výjimka* generovaná uvnitř procesoru (synchronní)

Interrupt odvozen od *externí* události (asynchronní)

Úkoly:

- *Detekce výjimky* – Jak odhalit výjimku
- *Ošetření výjimky* – Co dělat

MIPS: 2 typy – Nedefinované instrukce, aritmetické přetečení

Detekce výjimek

- **Nedefinované instrukce:**
  1. Doplnění stavu 10 [Exception] do FSC
  2. Každá instrukce vyjma lw, sw, beq, R-formát nebo jump způsobí přechod do stavu 10

- **Aritmetické přetečení (overflow):**
  1. Opakování: ALU obsahuje logiku detekce přetečení. Vytvoření dalšího stavu 11 v FSC pro obsluhu přetečení
  2. Vznikne-li *přetečení* (*Overflow) na výstupu* ALU, pak řadič přejde do stavu 11

Ošetření výjimek

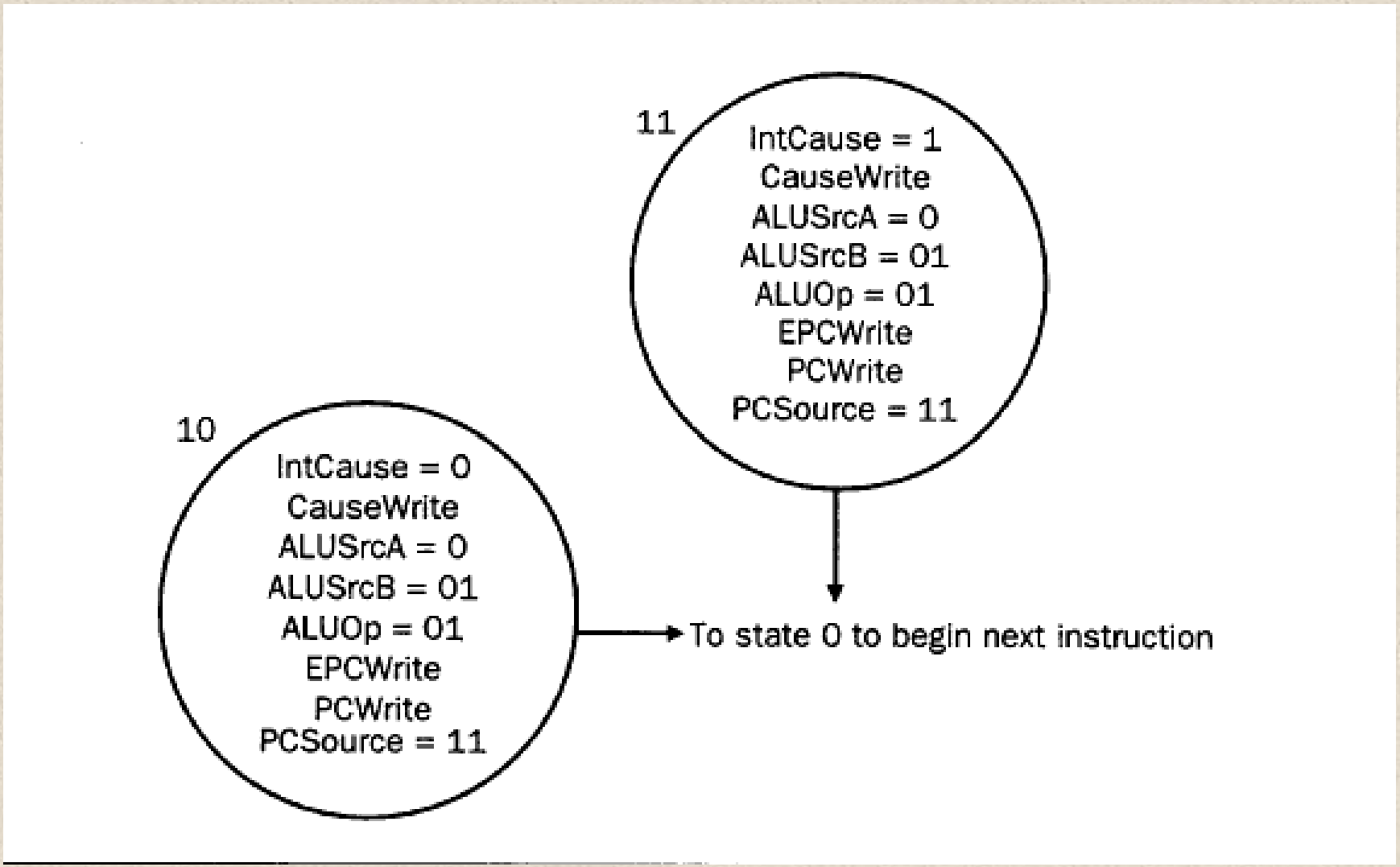
- **Dvě metody:** *EPC/Cause* a *Vektorové interrupty*
- *Vektorové interrupty*:
  1. Každá výjimka má vyhrazenou určitou adresu A_E
  2. Výjimka detekována => A_E je kvůli ošetření zapsána do PC
- *EPC / Příčina:* (MIPS) (EPC …Exception Program Counter)
  1. Výjimka detekována => Adresa instrukce uložena do **$epc**
     Cause registr obsahuje kód výjimky
  2. Obsluha výjimky vyšetří registr **Cause** a zkouší restartovat výpočet na místě, kam ukazuje **$epc.**

Úprava MIPS pro výjimky

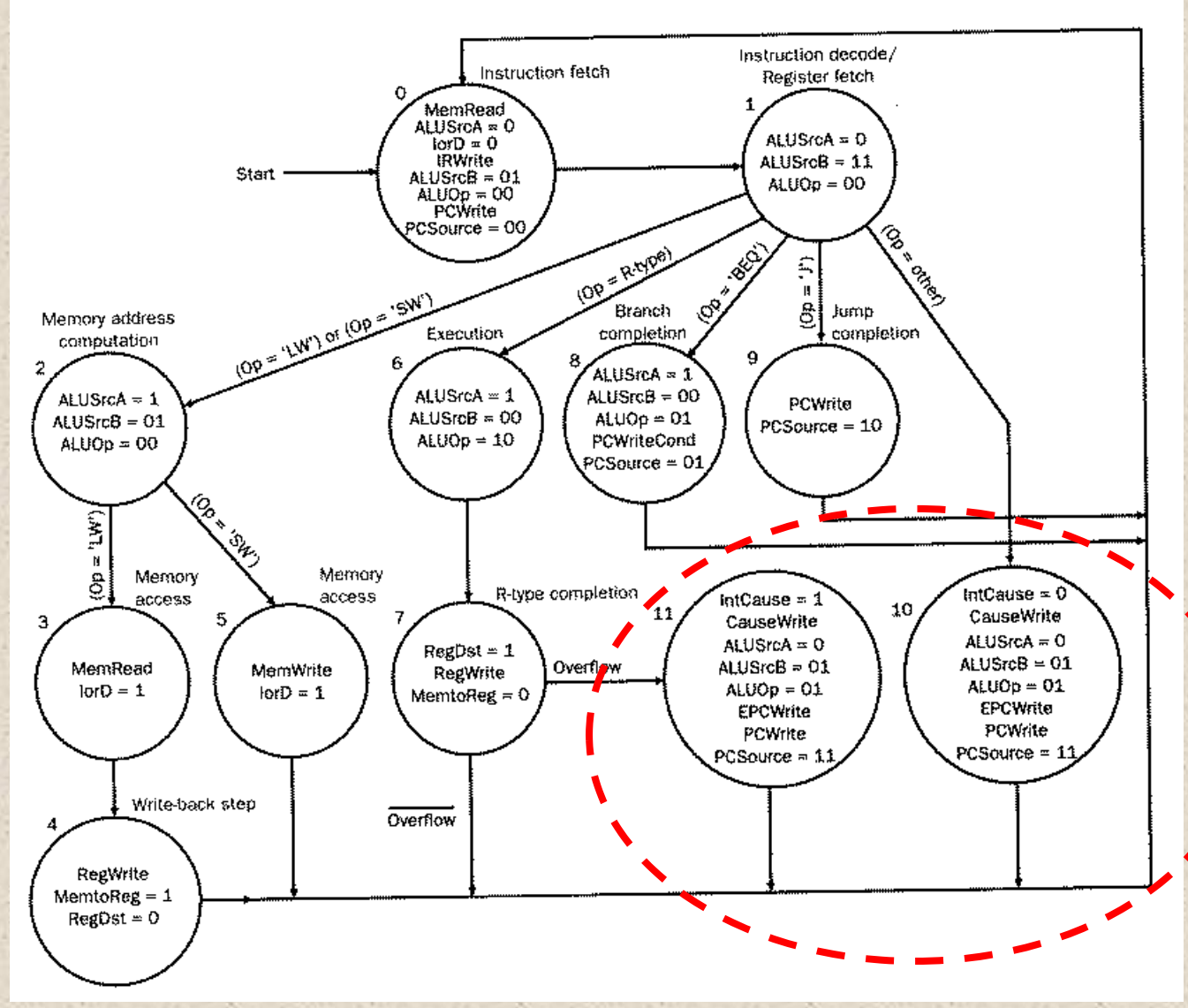
1. Nové registry - **$epc** a **Cause** (32-bitů)
2. Nové řídící signály – EpcWrite, CauseWrite
3. Nové řídící linky – 0 pro nedefinovanou instrukci
   1 pro přetečení
4. Nový Mux signál pro zdroj PC - PCsource = 11_2
   - Staré vstupy PC: PC+4, BTA, JTA
   - Nový vstup: $A_E = C0000000_{16}$ u MIPS

Opakování: detekce přetečení ALU "již instalováno"

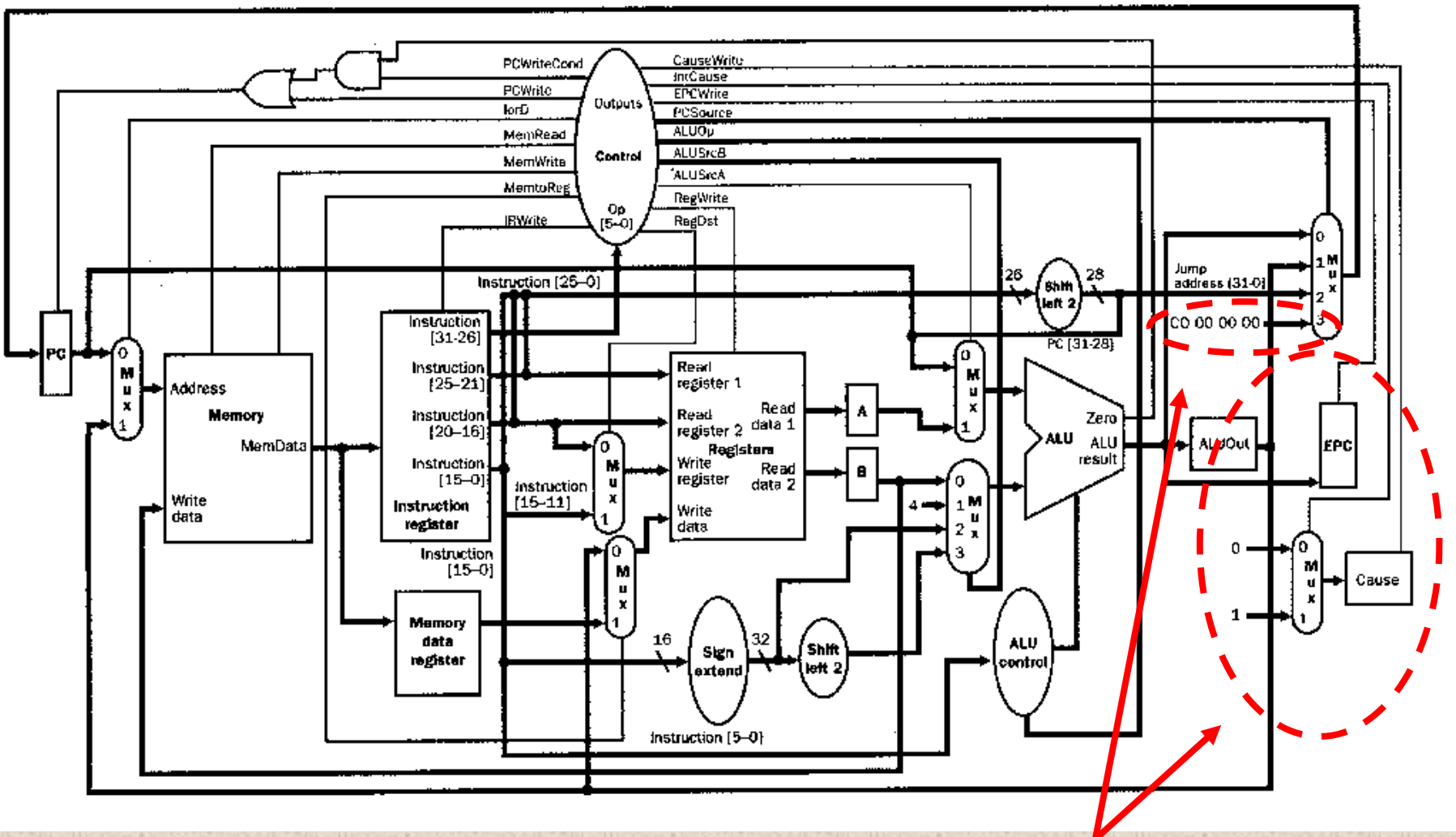
Nové stavy pro ošetření výjimek



Nový FSC se stavy výjimek



Nové uspořádání MIPS



HW pro ošetření výjimek

Problémy spojené s výjimkami

- **Rollback a restart**
  - *Rollback*: Reverzní proces
  - *Restart*: Opětné provedení procesu nebo operace
  - Detekce přetečení *po* zápisu výsledku ALUout
  - To znamená, že operace ALU způsobila neopravitelnou chybu ☹
  - Současný FSC nedovoluje podporu procesu *restart* nebo *rollback*

Jak to napravit?

- **Obtížné** – Návrh řídícího systému je komplikovaný
  – K provedení *rollback*u je třeba mnoho dalšího HW

Závěr

- **Mikroprogramování** – Flexibilnější pro rozlehlé ISA
- **Úkol:** Navrhnout pole a signály konzistentní
- **Omyl:** Mikroprogramy jsou nutně pomalejší
- **Omyl:** Přidávání instrukcí "je zdarma"
- **Výjimky** (interní události) vs. **Interrupty** (externí)
  - *Detekce výjimek* vyžaduje speciální hardware
  - *Ošetření výjimek* vyžaduje více hardware (cena!)

Závěr

- Jestliže rozumíme instrukcím…
  Můžeme postavit jednoduchý procesor!
- Trvají-li instrukce různou dobu, je výhodnější multicyklové zpracování
- Datové cesty se implementují s využitím:
  - Kombinační logiky pro aritmetiku
  - Paměťové prvky (klop. obvody) pro zapamatování informace
- Implementace řízení s využitím:
  - Kombinační logiky pro jednocyklovou implementaci
  - FSM pro vícecyklovou implementaci