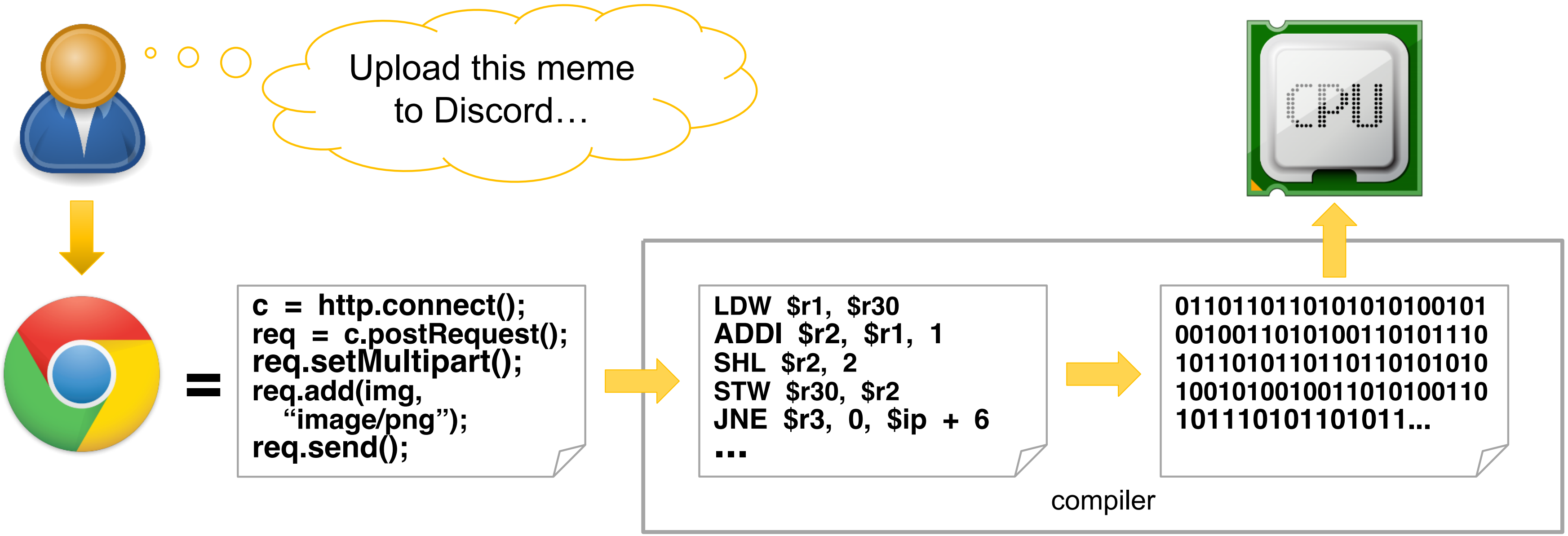
CPU

NSWI170 Computer Systems

Jakub Yaghob, Martin Kruliš

Semantic gap

- Gap between user intent and instructions executed by CPU



Von Neumann architecture

- System architecture
  - Program and data in the same memory
  - One shared bus
    - Evolution of the architecture

Je však HW jednodušší

Ale pomalejší, hlavně kvůli multi-jádrům

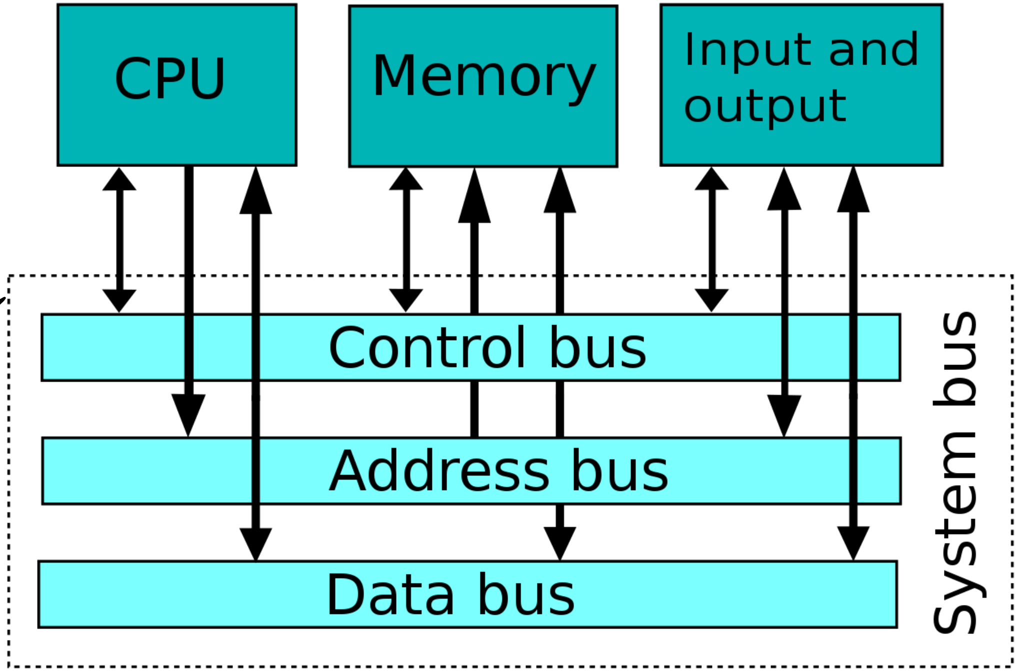
Jedna společná sdílená sběrnice

||

pomalé,

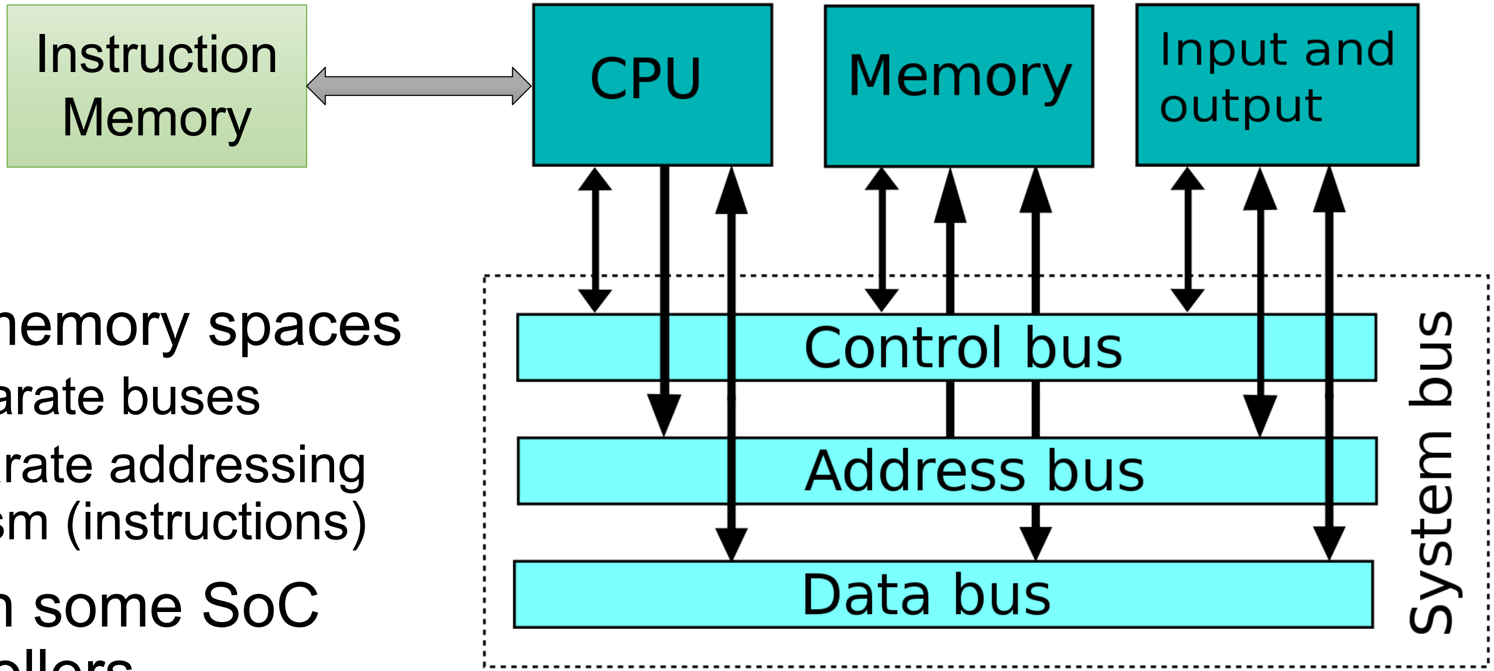
protože je

jeden exkluzivní přenos



Harvard architecture

→ Tohle je určení pro kód

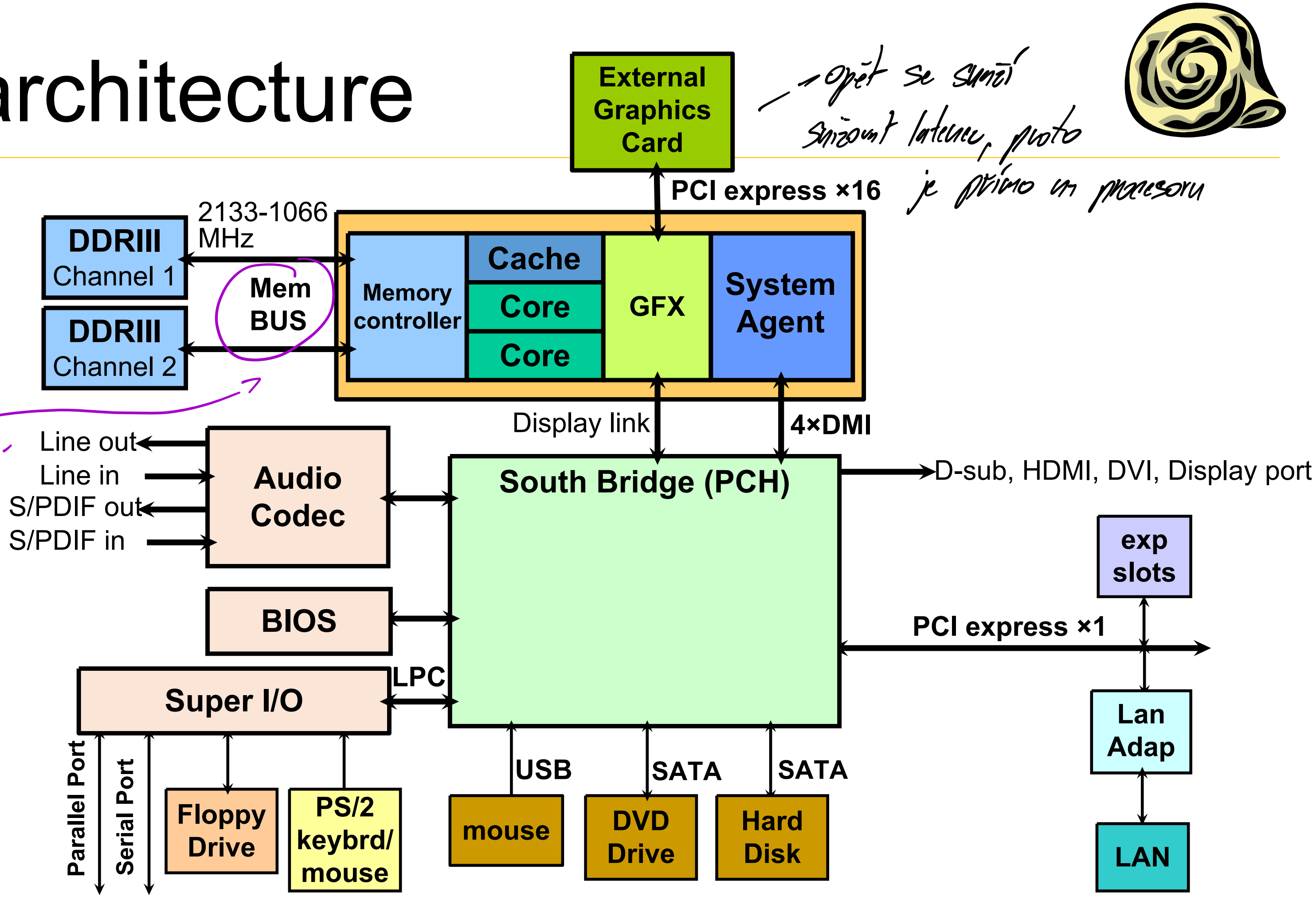


- Separate memory spaces
  - With separate buses
  - And separate addressing mechanism (instructions)
- Still used in some SoC microcontrollers

Většinou tam bývá ještě více datových prostorů

Real PC architecture

- Sandy Bridge



– opět se snaží snížit latenci, proto je přímo na procesoru

Vlastní paměťová sběrnice pro vyšší rychlost

Funguje se na sériové sběrnici, protože není přeslechy jiného vodiče

CPU

- CPU Architecture
  - Instruction Set Architecture (ISA) – Popisuje logické chování procesoru, říká co dělají které operace a co vyžadují
    - Abstract specification (contract between programmers and HW designers)
  - Hardware architecture
    - Actual implementation

  „Jak je to poskládané"
- What is CPU?
  - "Simple" machine that executes instructions
    - Instruction – simple command, arithmetic operation, …
  - Registers, memory controllers, I/O

x86 – nejpoužívanější
– zastaralý

Na základě ISA se dělá vlastní HW implementace.

Instruction

- Simple command to the CPU
  - Binary encoding (CPU), assembler (programmers)

MIPS32 Add Immediate Instruction

001000	00001	00010	0000000101011110
OP Code	Addr 1	Addr 2	Immediate value

Fixed vs. variable lengths

Equivalent mnemonic: **addi $r1, $r2, 350**

- Operands
  - Depending on ISA – max 1, 2, or 3 (some may be implicit)
  - Register, immediate value

Instruction cycle

- Sub-steps performed in every instruction
  - Load instruction from address stored in IP register
  - Decode the instruction
  - Load operands
  - Execute the operation
  - Store the result
  - Increment IP

Tohle jsou brzdy

Some steps may be skipped for some instructions

Instructions - motivation

- How can we execute the following code?

podmíněný nepodmíněný skok (abych přeskočil ten else)

```
if (a < 3) b = 4; else c = a << 2;
```

```
for (int i = 0; i < 5; ++i) a[i] = i;
```

```
int f(int p) { return p + 1; }
void g() { auto r = f(42); }
```

Instruction classes

- Load instructions → Pracují s pamětí => jsou pomalé
  - Memory -> register
  - Take a long time to execute, important to detect soon (fetch data ahead)
- Store instructions → Pracuje s pamětí => je nejpomalejší
  - Register/immediate -> memory
- Move instruction
  - Between registers → To je problémy load-store, move je ideální jen mezi registry
  - x86-64 also between registers and memory
    - Difficult to implement efficiently in HW

Instruction classes

- Arithmetic and logic instructions
  - **+, -, <<, >>, &, |, ^, ~**
  - ***, /, %**
- Jumps
  - Unconditional × conditional
  - Direct × indirect × relative
- Call, return
- …

No funny stuff like "**str**" *** 4**

ty musíme mít

Tests required (**==**, **<**, …)

přímo zadaná adresa

adresa uložena v paměti

přičítání offsetu

softwarově

Někde musí být implementovaný zásobník s návratovou adresou, kde jsem funkci volal.

Higher-level code structures

```
if  (a < 3) b = 4; else c = a << 2;
```

```
...
jge [a], 3
store [b], 4
jmp
load r1, [a]
shl r1, 2
store [c], r1
...
```

This is just a symbolic abstract assembler
(not an actual one)

Higher-level code structures

for (int i = 0; i < 5; ++i) a[i] = i;

```
mov r1, 0
jge r1, 5
load r2, [a]
add r2, r2, r1
store [r2], r1
add r1, r1, 1
jmp
...
```

Actually, the offset needs to be multiplied by **sizeof(a[0])**, but you got the point...

Registers jednotky – stovky

- Types
  - General, integer, floating point, bool
  - address, branch, flags, predicate,
  - application, system,
  - vector, …

True: vykonání tuto instrukci
False: nevykonání tuto instrukci
nemusí se skákat

- Naming
  - Direct (EAX, r01, …) × stack (relative addressing to top of the stack)
- Aliasing

Registers – example 32-bit x86

Obecné registry $\overbrace{\text{AH} \quad \text{AL}}^{\bullet X}$

EAX		AX	AH	AL	akumulátor
EBX		BX	BH	BL	base
ECX		CX	CH	CL	count
EDX		DX	DH	DL	data
ESI			SI		source
EDI			DI		destination
EBP			BP		base pointer
ESP			SP		stack pointer

→ Segmentové registry

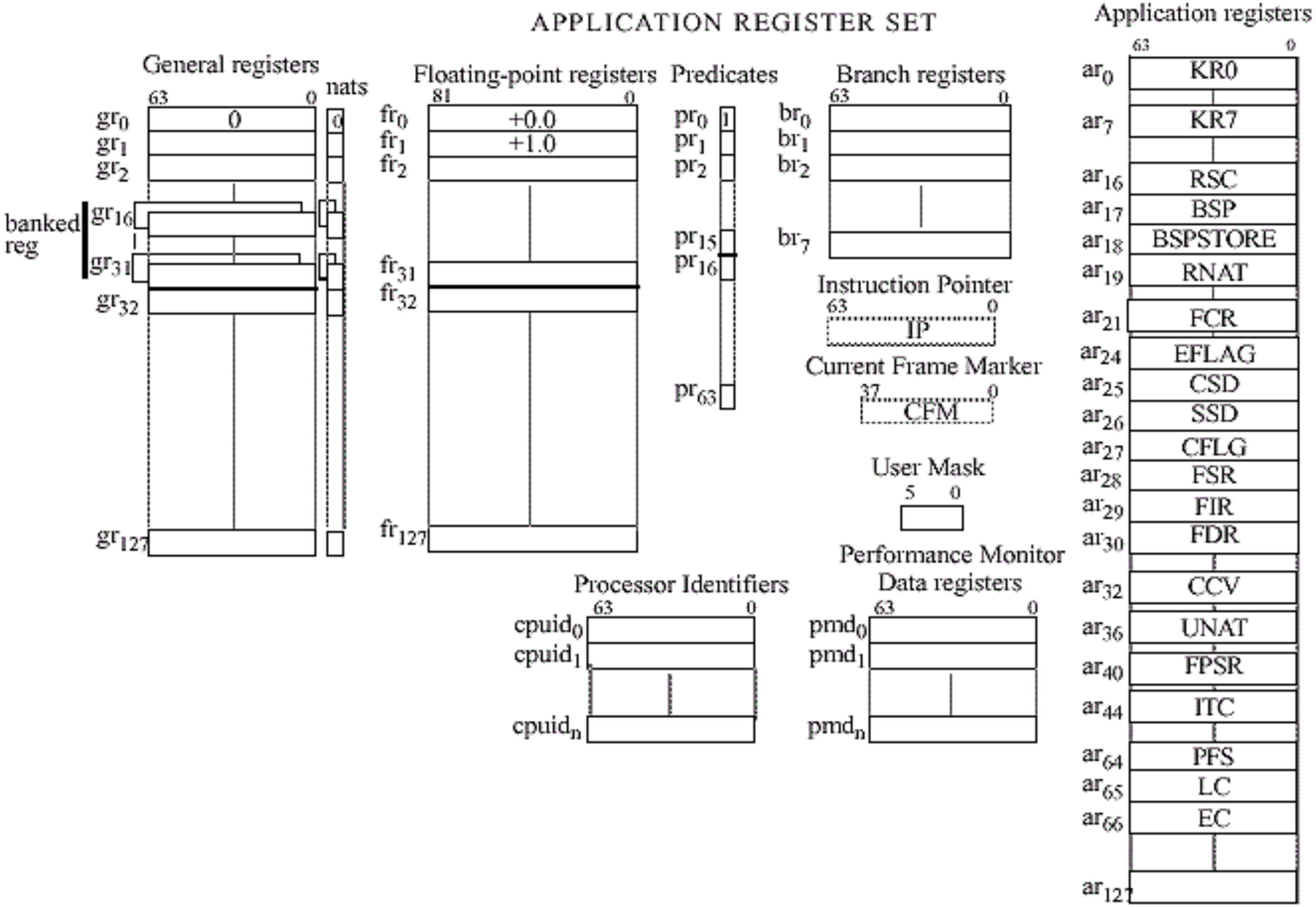
CS		
DS		
ES		
SS		
FS		
GS		
EFLAGS		FLAGS
EIP		IP

→ instruction pointer

x86-64 extends registers to 64bits (RAX, RBX, …)
+ adds general regs. R8 - R15

nejsou ortogonální → registry jsou spojení s některými instrukcemi

Registers – example IA-64



MIPS – simple assembler

- Execution environment
  - 32-bit (almost general) registers r0-r31
    - r0 is always 0, writes are ignored –> běžná věc na spoustě architektur
    - r31 is a link register for the **jal** instruction
  - No stack
    - Realized by SW
  - No flags
    - Slightly different instruction set (especially tests and conditional jumps)
  - Program Counter (PC) register

Application Binary Interface

- Application Binary Interface (ABI) ~ předpis „jak se to má chovat"
  - Additional specification that accompanies ISA
  - Prescribes how the CPU should be used
    - Important for compiler designers
    - If application and a library are compiled separately, they need to be compatible
  - Binary interfaces between applications
    - Typical utilization (meaning) of registers
    - Stack implementation and layout
    - Function call conventions
    - …

MIPS – register aliases

Tohle specifikuje ABI

Register	Name	Purpose	Preserve
$r0	$zero	0	N/A
$r1	$at	Assembler temporary	No
$r2-$r3	$v0-$v1	Return value	No
$r4-$r7	$a0-$a3	Function arguments	No
$r8-$r15	$t0-$t7	Temporaries	No
$r16-$r23	$s0-$s7	Saved temporaries – garantována stabilita	Yes
$r24-$r25	$t8-$t9	Temporaries	No
$r26-$r27	$k0-$k1	Kernel registers – DO NOT USE	N/A
$r28	$gp	Global pointer	Yes
$r29	$sp	Stack pointer	Yes
$r30	$fp	Frame pointer	Yes
$r31	$ra	Return address	Yes

Důležité

ostatní funkce to musí uložit bokem a pak zase vrátit →

MIPS – instructions

- Arithmetic
  - **add $rd,$rs,$rt**
    - R[rd] = R[rs] + R[rt]
  - **addi $rd,$rs,imm16**
    - R[rd] = R[rs] + signext(imm16)
  - **sub $rd,$rs,$rt**
  - **subi $rd,$rs,imm16**

These are only symbolic placeholders, real example could look like **add $r2,$r4,$r5**

Immediate value, 16 bit number encoded directly in the instruction itself

Signed extension to 32bits (to match size of the registers)

ISA comparison

MIPS

add $t1,$t1,$t0
addi $t1,$t1,1

add $t2,$t0,$t1

Tříadresová instrukční sada je vždy výhodou

x86

tady jde jen eax += ebx

add eax,ebx
add eax,1
or **inc eax**

mov eax,ebx
add eax,ecx

MIPS – instructions

- Logic operations
  - **and $rd,$rs,$rt** **andi $rd,$rs,imm16**
  - **or $rd,$rs,$rt** **ori $rd,$rs,imm16**
  - **xor $rd,$rs,$rt** **xori $rd,$rs,imm16**
  - **nor $rd,$rs,$rt**
    - R[rd] = R[rs] and/or/xor zeroext (imm16)
  - No **not** instruction, use **nor $rd,$rs,$rs**
- Shifts
  - **sll** / **slr $rd,$rs,shamt**
    - R[rd] = R[rs] << / >> shamt
  - **sra $rd,$rs,shamt**

Arithmetic shift (keeps the sign)

ISA comparison

MIPS

```
nor  $t1,$t2,$t2

sll  $t1,$t1,3
```

x86

```
mov  eax,ebx
not  eax

shl  eax,3
```

MIPS – instructions

MIPS má všechny registry 32bit only. (Není aliasing)

- Memory access

  vezmu si 32bitů do registru

  „load word"
  - **lw $rd,imm16($rs)**
    - R[rd] = M[R[rs] + signext32(imm16)]

  „store word"
  - **sw $rt,imm16($rs)**
    - M[R[rs] + signext32(imm16)] = R[rt]

  „load byte"
  - **lb $rd,imm16($rs)**
    - R[rd] = signext32(M[R[rs] + signext32 (imm16)])
  - **lbu $rd,imm16($rs)**
    - R[rd] = zeroext32(M[R[rs] + signext32 (imm16)])

  „store byte"
  - **sb $rt,imm16($rs)**
    - M[R[rs] + signext32(imm16)] = R[rt]

- Moves

  „load immediate"
  - **li $rd,imm32**
    - R[rd] = imm32
    - Pseudo-instruction, translates to **lui** and **ori**
      - Load upper immediate
  - **move $rd,$rs**
    - R[rd] = R[rs]

ISA comparison

MIPS

```
lw     $t1,1234($t0)
sw     $t1,1234($t0)
lb     $t1,1234($t0)
li     $t1,5678
move   $t1,$t0
```

x86

```
mov  eax,[ebx+1234]
mov  [ebx+1234],eax
mov  al,[ebx+1234]
mov  eax,5678
mov  eax,ebx
```

MIPS – instructions

- Jumps
  - **j addr** → přímý
    - PC = addr
  - **jr $rs** → nepřímý
    - PC = R[rs]
  - **jal addr**
    - "Jump and link"
    - R[31] = PC+4
    - PC = addr

jal instruction takes up 4 bytes

ISA comparison

MIPS

```
    j      label
    jr     $ra

label:
    jal  fnc
```

Placeholder that marks a place in the code, the actual address is computed by a compiler

x86

```
    jmp  label
    jmp  [ebx]

label:
    call  fnc
```

This is double indirection!!! Register **ebx** holds address where is the pointer that is loaded and used as target address for the jump!

Return address stored to stack!

MIPS – instructions

- Conditional jumps
  - **beq $rs,$rt,addr** → rovnost
    - If R[rs] == R[rt] then PC = addr else PC = PC+4
  - **bne $rs,$rt,addr**
    - Analogical (not equal) → nerovnost
- Testing
  - **slt** / **sltu $rd,$rs,$rt**
    - If R[rs]<R[rt] then R[rd] = 1 else R[rd] = 0
  - **slti** / **sltiu $rd,$rs,imm16**
    - If R[rs] < signext/zeroext(imm16) then R[rd] = 1 else R[rd] = 0

sltu is unsigned version of **slt**

Only lesser-than test, greater-than is created by swapping operands

Přestože nejsou vyjádřeny všechny porovnání, umíme je nepřímo nebo dvojdobě vyřešit.

ISA comparison

MIPS

```
beq  $t0,$t1,label
```

```
slt  $t2,$t1,$t0
bne  $t2,$zero,label
```

```
slti  $t2,$t1,5
bne  $t2,$zero,label
```

x86

```
cmp  eax,ebx
jz   label
```

Universal compare (result stored in flags)

Jump decision based on flags

```
cmp  eax,ebx
jl  label
```

```
cmp  eax,5
jl  label
```

Code examples

- Simple for-loop

```
for  (int  i  =  0;  i  <  N;  i++)
{
   A[i]  =  42;
}
```

```
      addi   $t0, $gp, 28         # $t0 <- address of A
      lw     $t1, 4($gp)          # fetch N
      move   $t2, $zero           # i = 0
      ori    $t3, $zero, 42       # $t3 = 42
      j      cond                 # go to condition
body:
      sll    $t4, $t2, 2          # i*4 -> offset
      add    $t4, $t4, $t0        # A+i*4
      sw     $t3, 0($t4)          # A[i] = 42
      addi   $t2, $t2, 1          # i = i+1
cond:
      slt    $t4, $t2, $t1        # are we there yet?
      bne    $t4, $zero, body     # no, we're NOT done
```

Code examples

i = N*5+3;

```
lw      $t0,  4($gp)        #  fetch  N
ori     $t1,  $zero,  5     #  5
mult    $t0,  $t0,  $t1     #  N*5(fake)
addi    $t0,  $t0,  3       #  N*5 + 3
sw      $t0,  0($gp)        #  i = ...
```

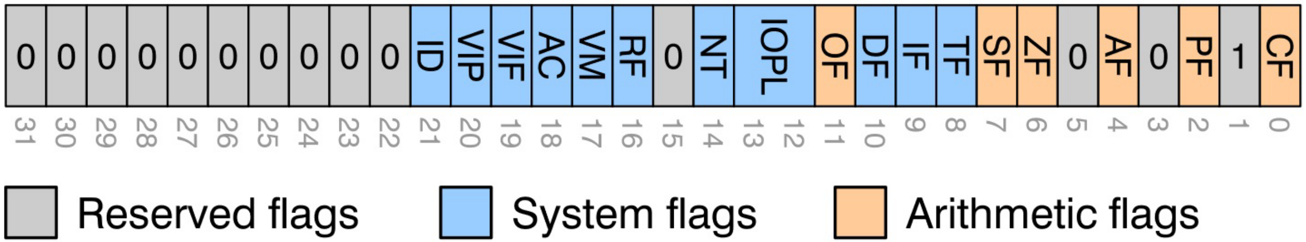
```
lw      $t0,  4($gp)        #  fetch  N
sll     $t1,  t0,  2        #  N*4
add     $t1,  $t1,  t0      #  N*4+N
addi    $t1,  $t1,  3       #  N+3
sw      $t1,  0($gp)        #  i = ...
```

Code can be optimized…

Flags

- Only used by some ISA
- Control execution
- Check status of the last instruction
- Usual flags
  - Z – zero flag
  - S – sign flag
  - C – carry flag

x86 example



ISA consolidation

ARM býval RISC, už je ale moc komplexní
x86 býval CISC, nyní se posouvá do RICS
- ty složité jsou jen kvůli zpětné kompatibilitě

- Instruction set architecture
  - Abstract model of CPU
- Classification
  - CISC – Complex Instruction Set Computer
  - RISC – Reduced Instruction Set Computer
  - VLIW – Very Long Instruction Word → není žádný dekodér, aby to bylo rychlejší
  - EPIC – Explicitly Parallel Instruction Computer
    ↳ zakódování co lze počítat paralelně.
- Orthogonality
  - Accumulator
- Load-Execute-Store → ARM, MIPS ...
  Jen x86 nemá dost registrů...

Tohle ale nebrzdí execution, jen loading/store

Hardware architecture

- CPU comprise…
  - Memory controller — kvůli vyšší rychlosti paměti
  - Cache hierarchy — L1_3 … různé rychlosti, velikosti
  - Core
    - Registers
      - Types
    - Logical processor
      - Hyper threading

CPU – simplified scheme

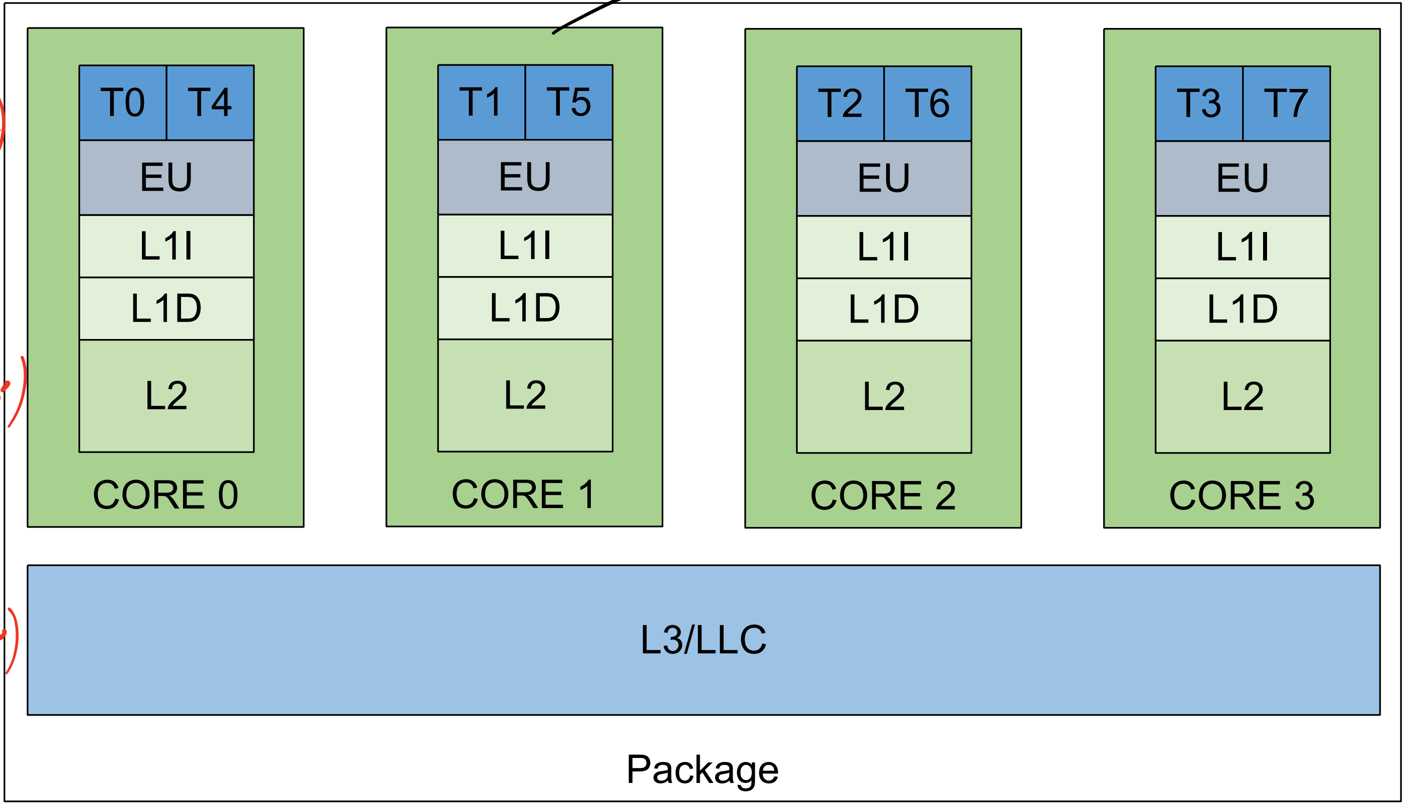
data instruction

L1 je nejrychlejší (4 takty)

Threads – vlákna

(16 taktů)

(100 taktů)



Threads:

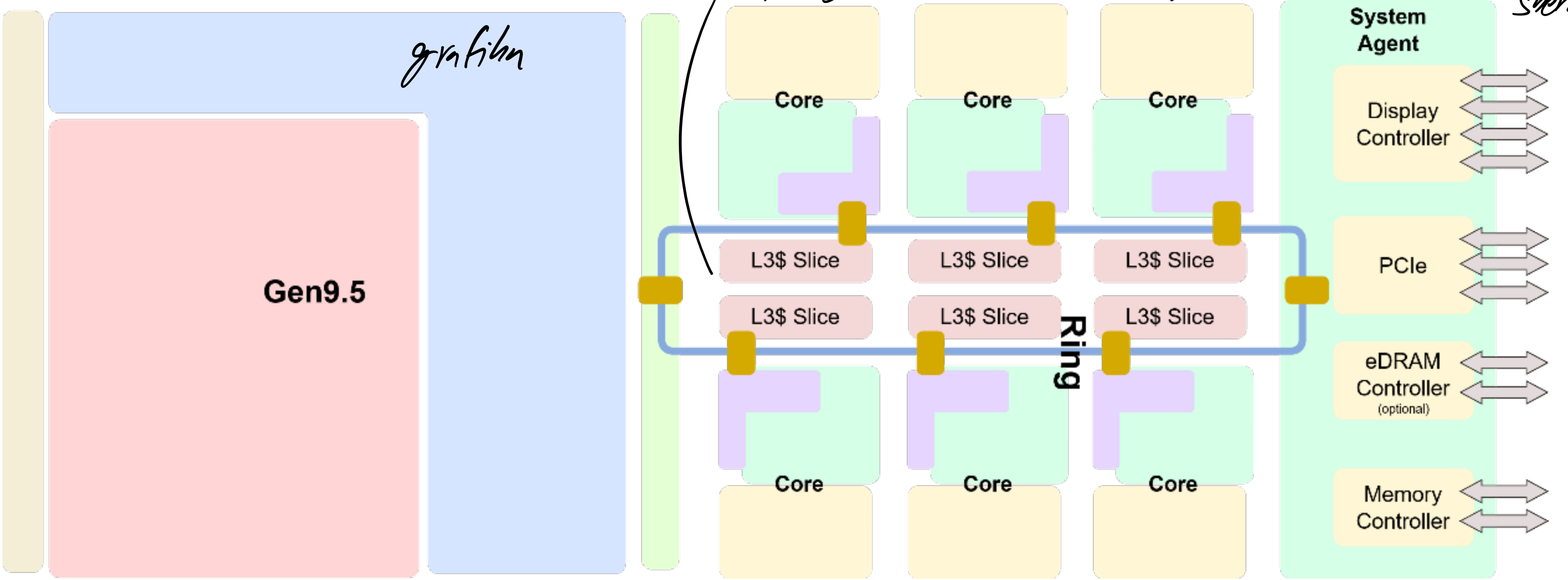
Jednotky nejsou tolik vytížený, takže přidám ještě jedno vlákno. +20 až 50%

(výpočetní jednotka je sdílená)

Scheme of Real CPU (Package)

- Intel Coffee Lake

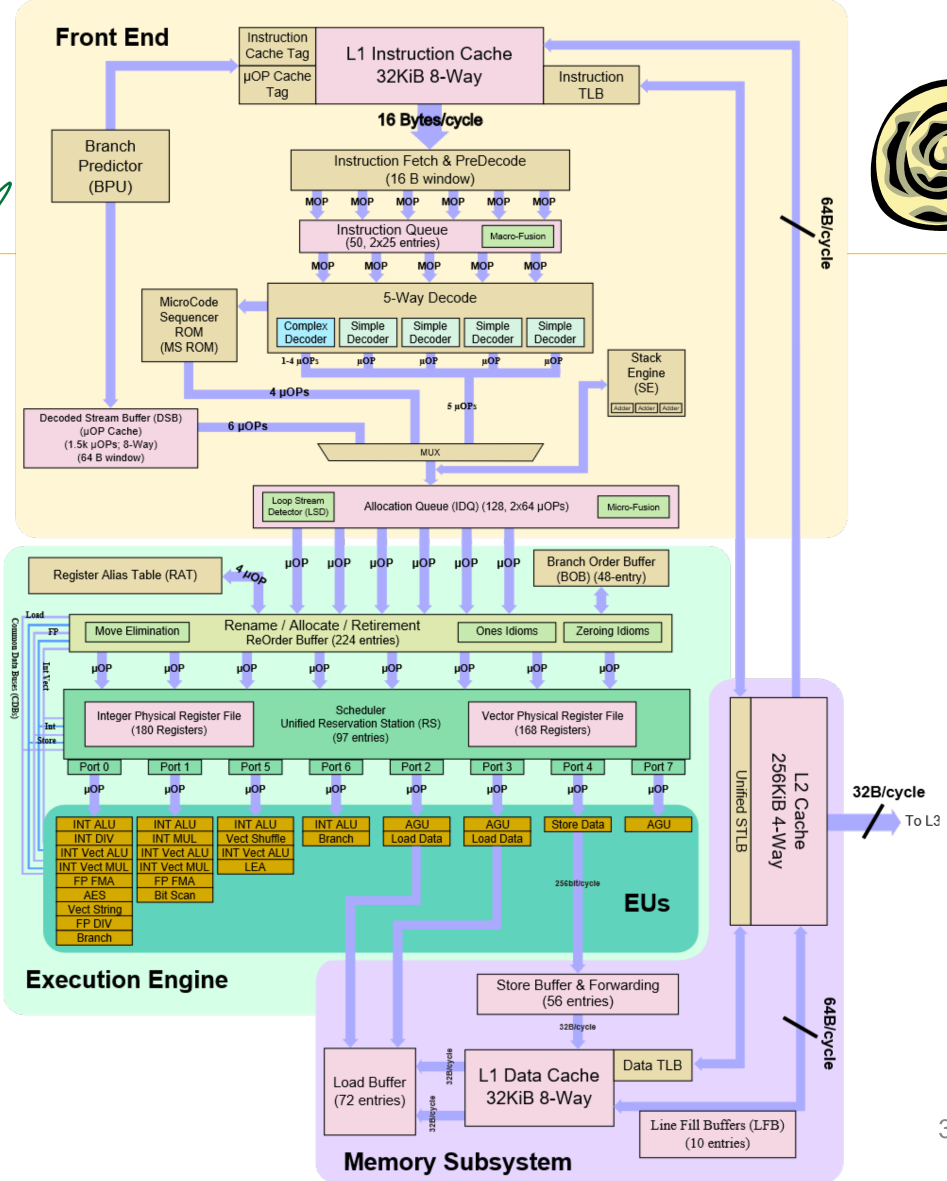
každý jádro má jen část L3 cache, takže to chci držet co nejvíc lokálně, abych nevyužíval sběrnici



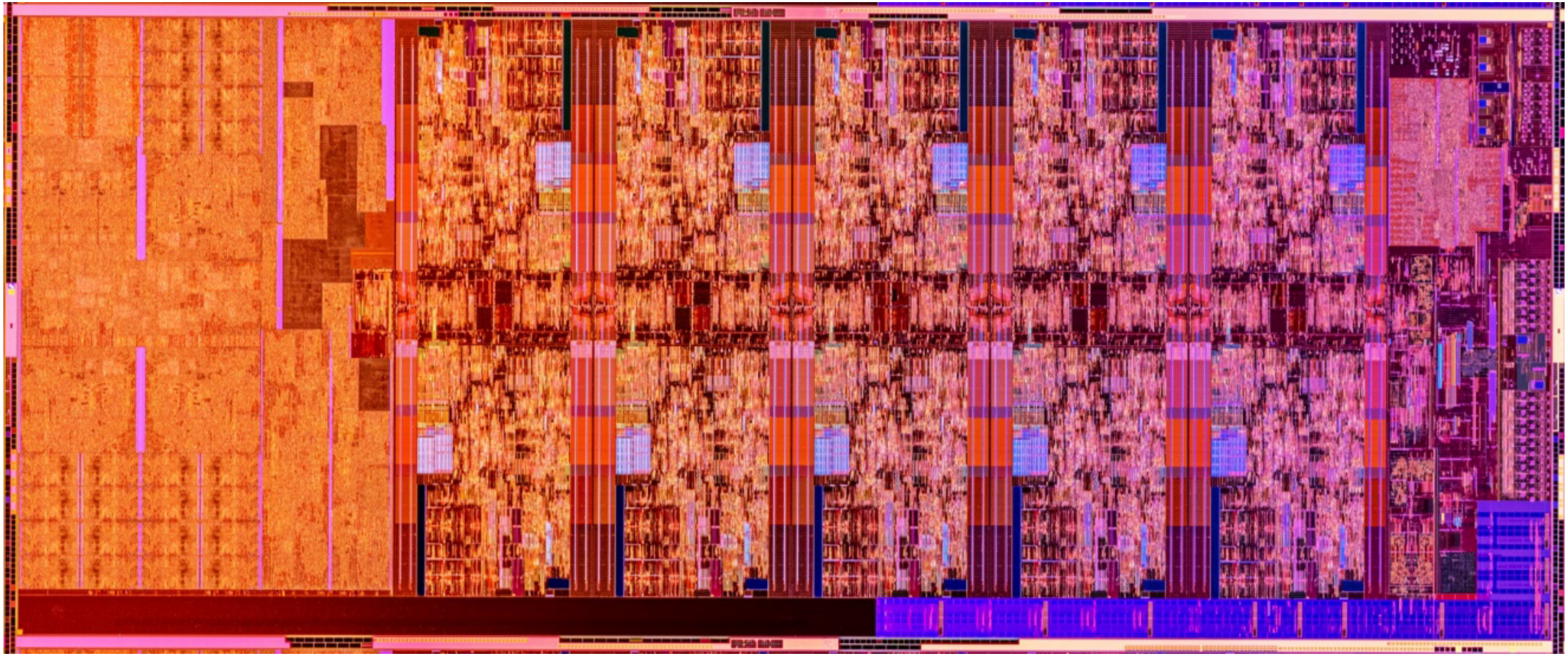
Real CPU

- Scheme of one core

96% úspěšnost



Real CPU Die



CPU architecture – pipeline

- Current CPU – většinou jeden dobu je jeden „takt".
  - 14-19 stages

Může docházet ke kolizím

Nejhorší, co je, je při pipeline, protože pak to musím spustit znova

Za jednotku času toho zvládnu víc

fetch	decode	execute	memory read	write back				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

– celá instrukce má 5 stages cca

i t

Náběh pipeline, nevyužito

identický výběh

CPU architecture – superscalar processor

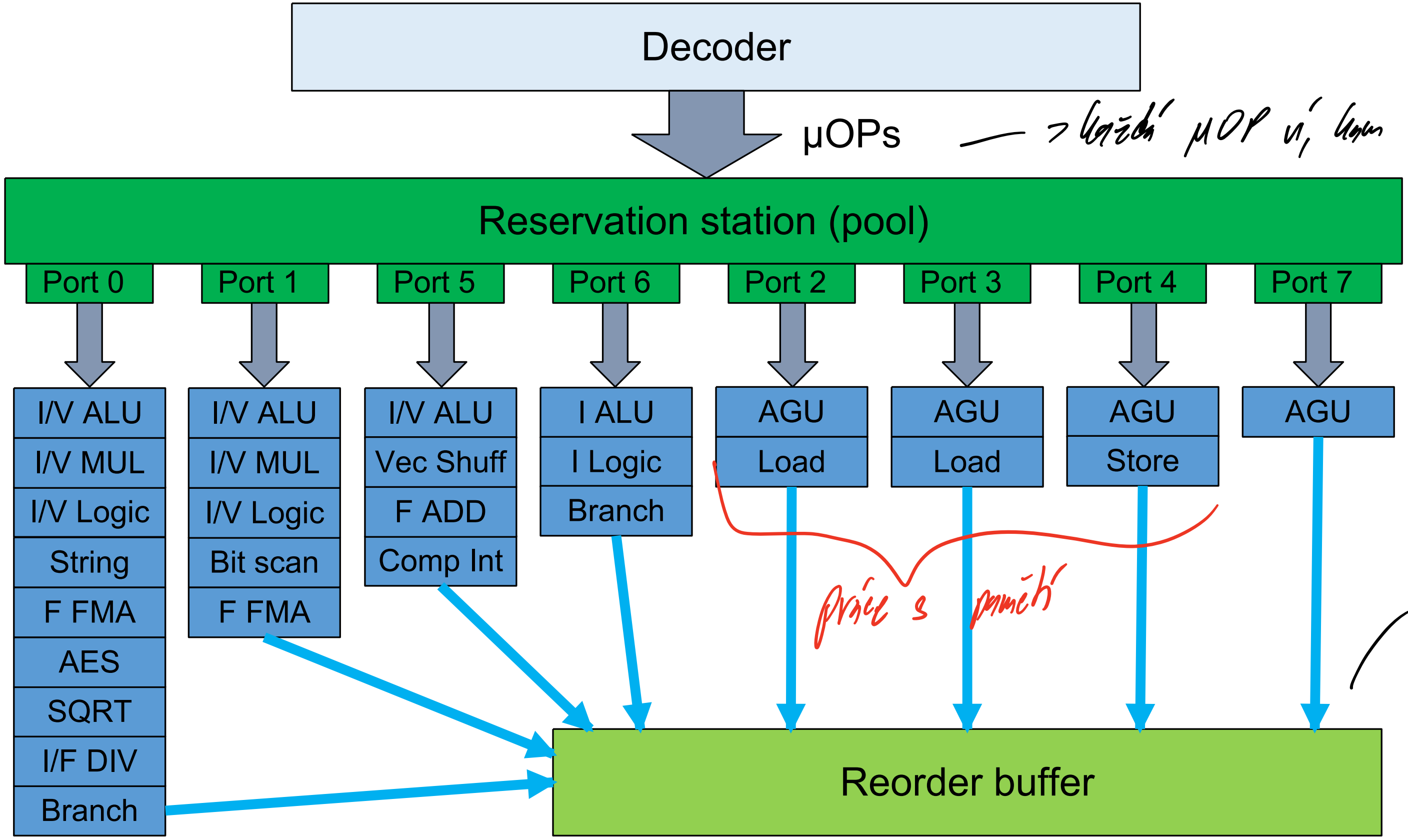
- Current CPU → Můžu mít víc instrukcí najednou
  - 5-way, asymmetric

Tady hodně bolí špatně odhadnutý skok, jelikož toho už hodně zahazuju.

IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

i ↓ t →

CPU architecture – out-of-order execution



→ každý µOP ví, kam chce jít

→ Výsledek se pak spojí do správného sekvenčního pořadí.

Discussion