

Microcode: x86 assembly is high level

Federico Cerutti / ceres-c

2024-10-21

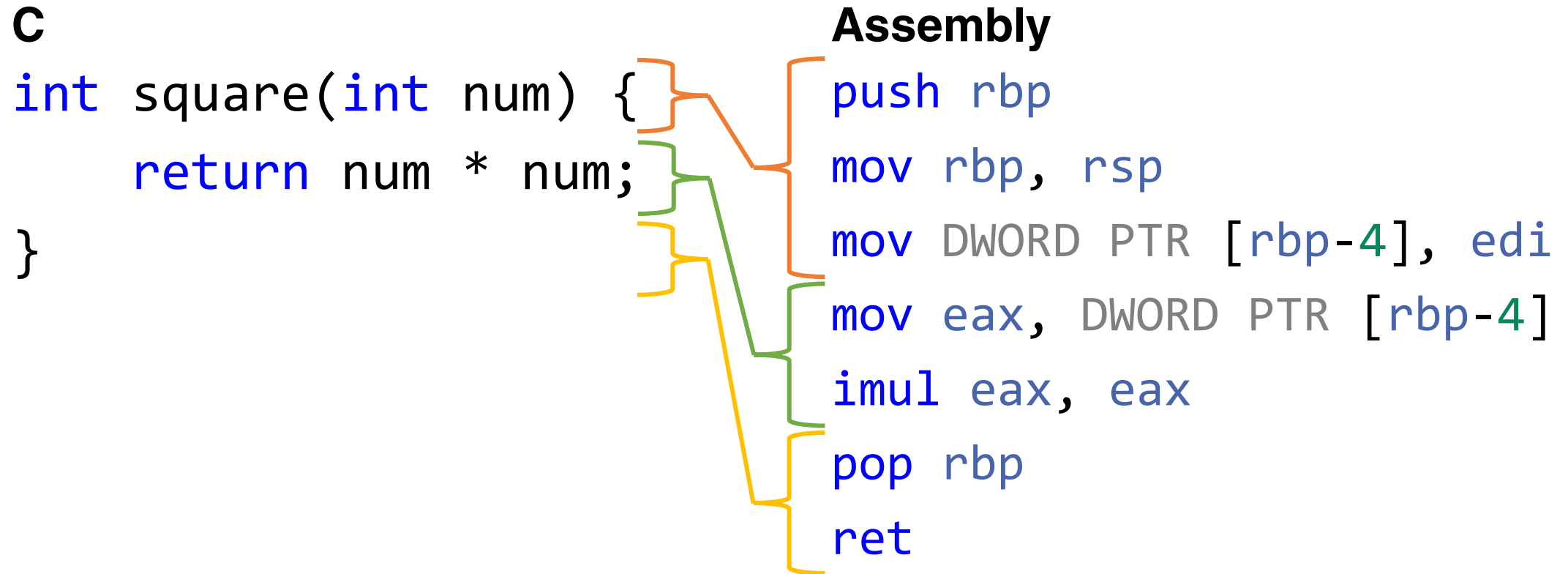
Roadmap

1. Assembly
2. Come funziona una CPU
3. Microcode
4. CPU debugging
5. Voltage glitching *bonus*

Assembly

C vs Assembly (x86)

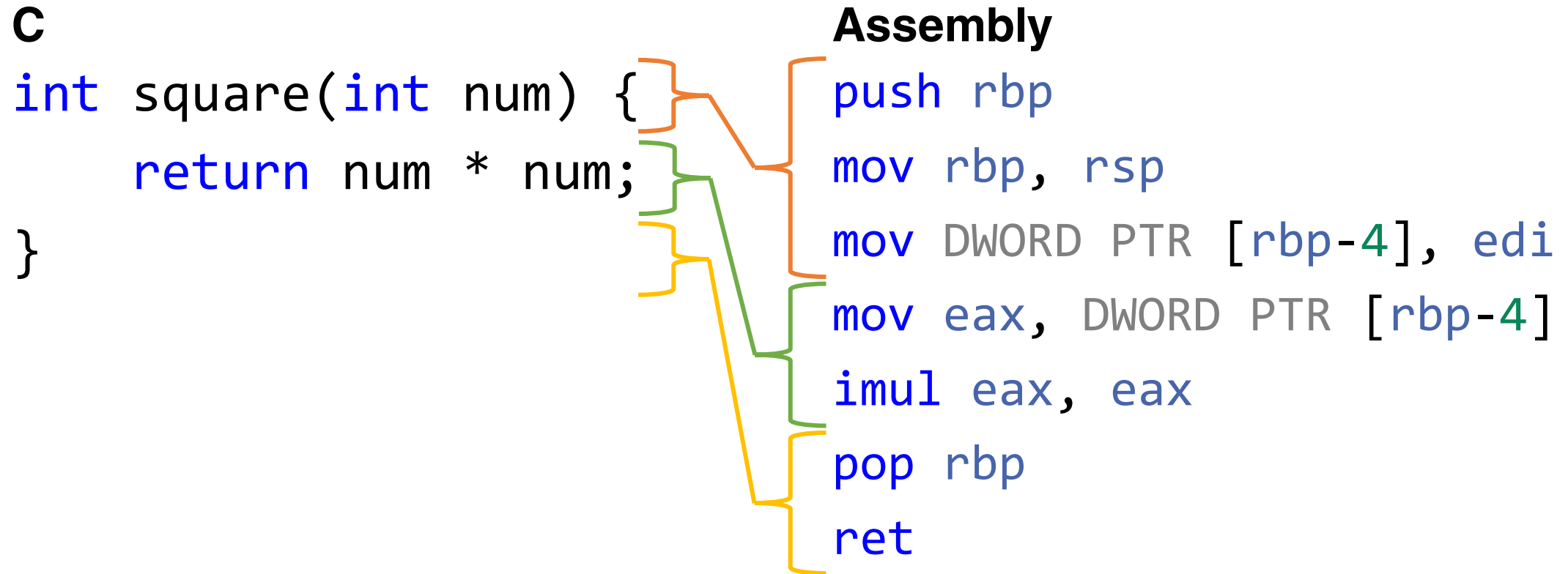
«Assembly è il linguaggio di programmazione a più basso livello»



C vs Assembly (x86)

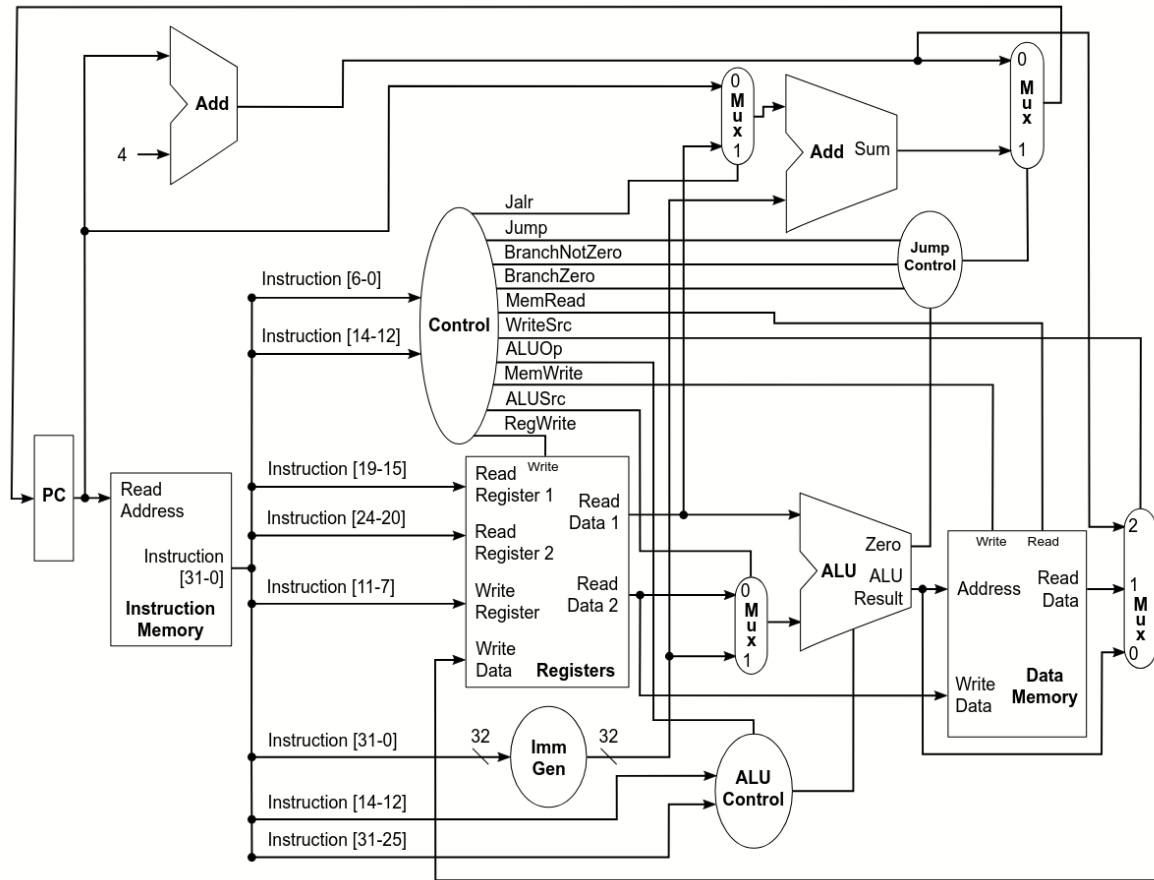
~~«Assembly è il linguaggio di programmazione a più basso livello»~~

FALSO



Come funziona una CPU

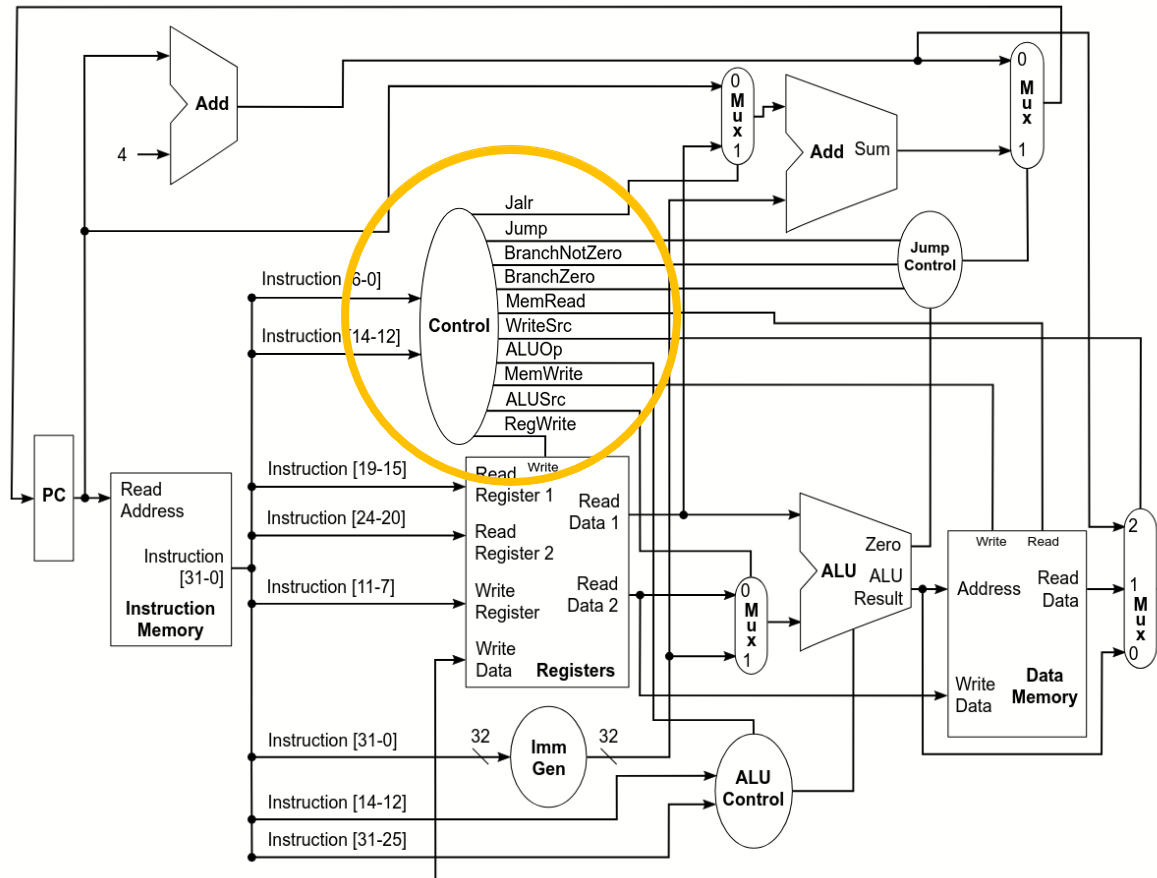
Textbook datapath



Patterson, Hennessy - Computer Organization and Design

ceres-c - Microcode: x86 asm is high level

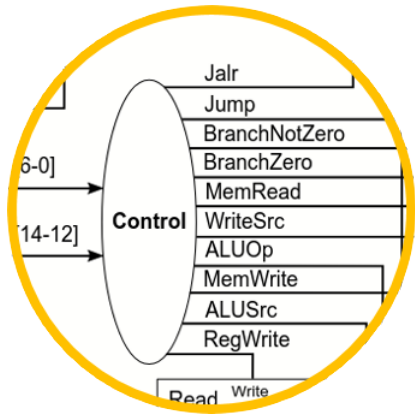
Textbook datapath



Patterson, Hennessy - Computer Organization and Design

ceres-c - Microcode: x86 asm is high level

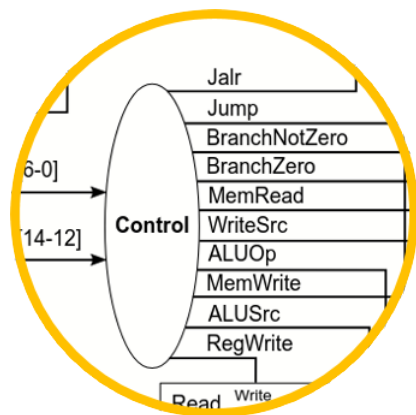
Control Unit



- **Orchestra** le operazioni
- Genera **segnali di controllo** per le componenti della CPU
 - **Decoder**: bit istruzioni assembly -> segnali

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
ALUOp0	0	0	0	1	

Control Unit



- **Orchestra** le operazioni
- Genera **segnali di controllo** per le componenti della CPU
 - **Decoder**: bit istruzioni assembly -> segnali
- Ok finché non esistono opcode come
 - CPUID
 - VMENTER/VMEXIT

Intermezzo: RISC vs CISC

Reduced Instruction Set Computer

- Poche istruzioni
- Data type semplici (byte/int)
- Esecuzione costant time

Complex Instruction Set Computer

- Molte istruzioni
- Data type complessi (vettori)
 - Diversi addressing mode
- Esecuzione variable time

x86 (CISC)

- **Molte** istruzioni (1500+)
 - Ogni istruzione richiede **transistor**
 - I transistor **costano**
- Alcune istruzioni **poco usate**
 - e.g. CPUID
 - Aumento **costi** non giustificabile

x86 (CISC)

- **Molte** istruzioni (1500+)
 - Ogni istruzione richiede **transistor**
 - I transistor **costano**
- Alcune istruzioni **poco usate**
 - e.g. CPUID
 - Aumento **costi** non giustificabile

- Soluzione: **Microcode**

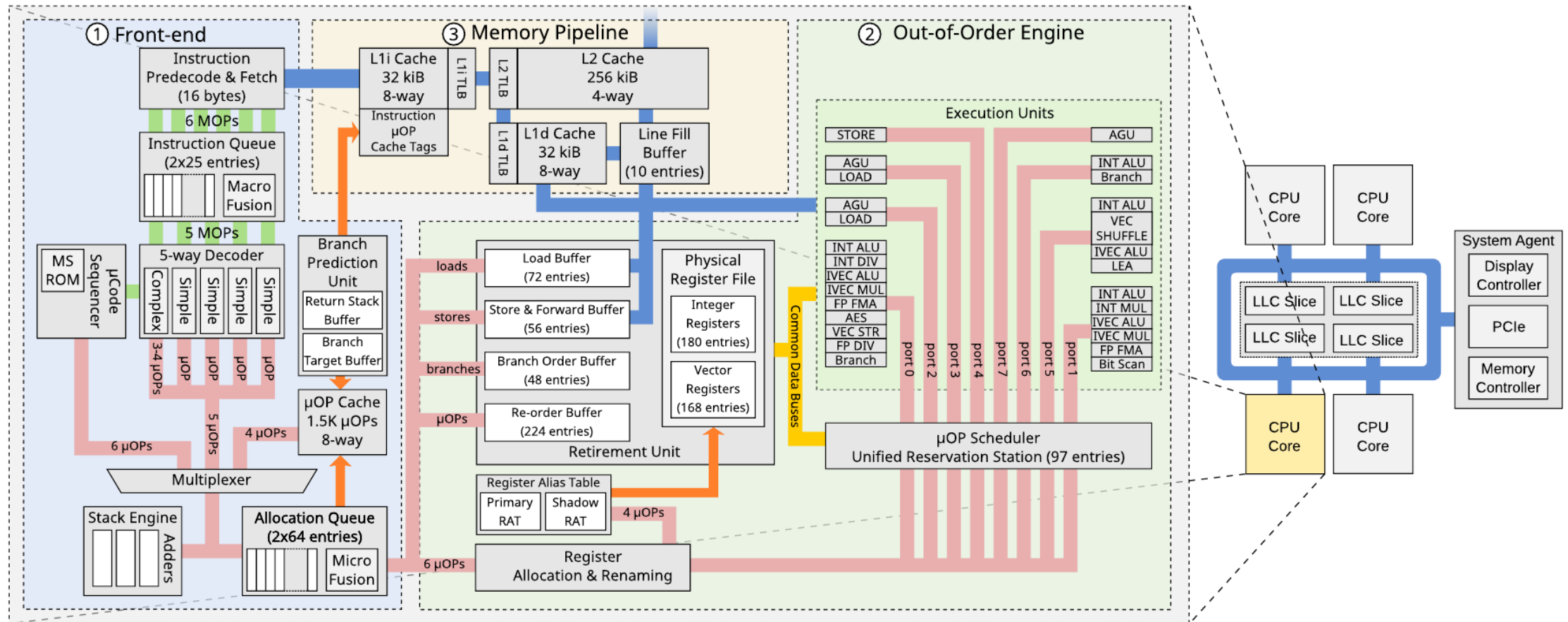
Microcode

Microcode

- La CPU è **RISC***, ma assembly **CISC**
 - Assembly x86 tradotto in un instruction set più semplice
 - Istruzioni x86 implementate da un **microprogrammi**
- Riuso hardware ⇒ Meno transistor
 - **Costi** produzione inferiori
 - **Consumo energetico** inferiore
- **Aggiornabile** (bugfix)
 - Aggiornamenti **criptati e non documentati**

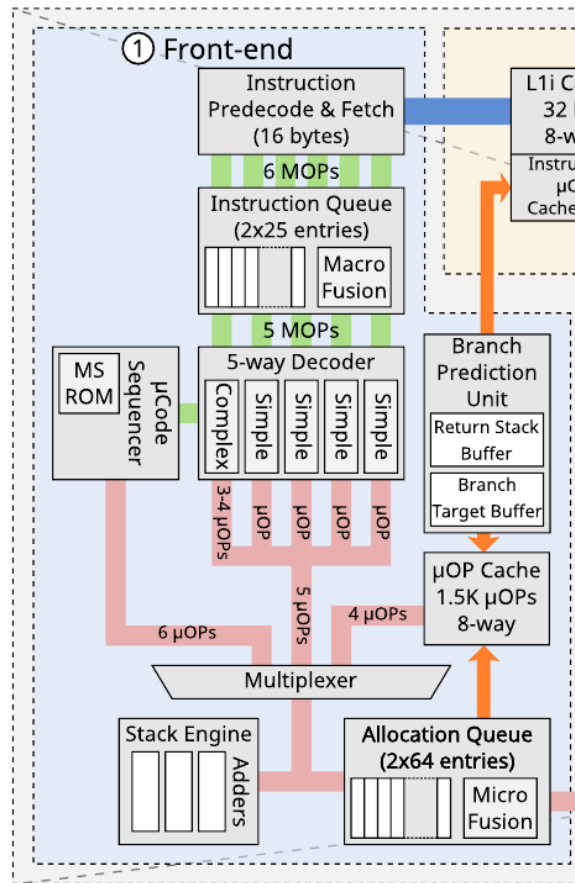
*terms and conditions may apply

Come funziona (davvero) una CPU



Borrello, Schwarzl – Custom Processing Unit

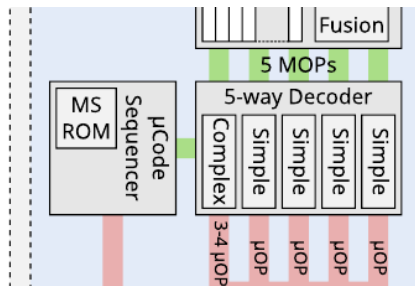
Frontend: Flusso Istruzioni



1. Fetch
2. Cache
- 2a. Instruction fusion
3. Decode:
 - a) Simple decoder $1 \text{ opcode} = 1 \mu\text{-op}$
 - b) Complex decoder $1 \text{ opcode} = 3\text{-}4 \mu\text{-op}$
 - c) Microcode sequencer $1 \text{ opcode} = n \mu\text{-op}$
4. Execution queue...

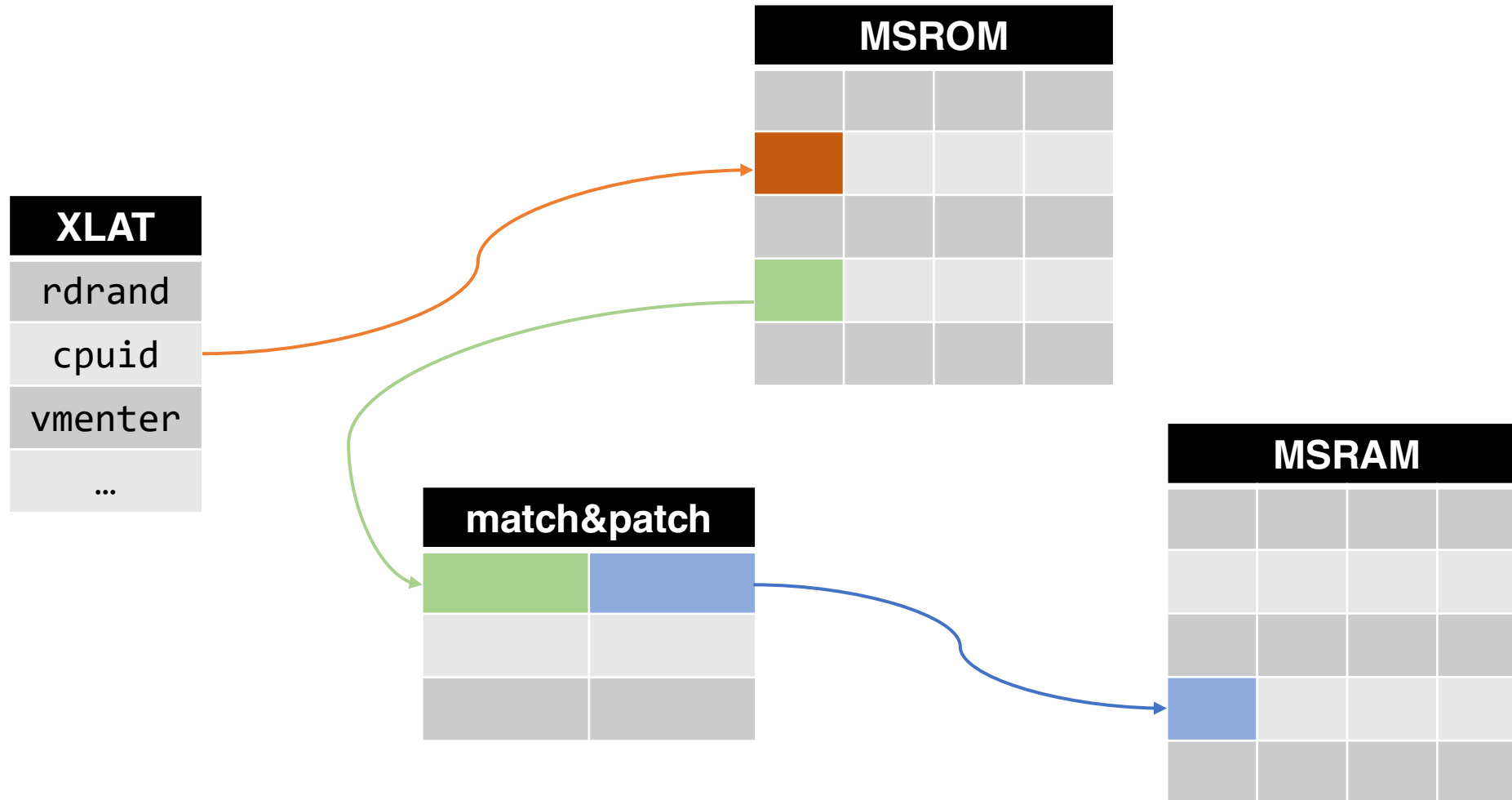
We pretend speculative execution does not exist

Microcode Sequencer



- Implementa le istruzioni **più complesse**
 - **Non esegue** direttamente μ -op
- Sembra un **processore**, ha:
 - Registri
 - RAM
 - ROM (programmi)
 - Instruction Pointer
 - Accesso a periferiche
- Componente **aggiornabile** del microcode
Solo aggiornamenti **firmati**

Microcode Sequencer: translation



CPU Debugging

Intel Debugging

I prodotti Intel dispongono di **avanzate funzionalità di debug**.

Accesso allo **stato interno** delle periferiche nella CPU
⇒ data leak, code execution, security bypass **✗**

Sistema controllo accessi: **✓**

- **Red** Unlock: Intel **internal**
- **Orange** Unlock: BIOS vendors
- **Green** Unlock: Customers

Intel Debugging

I prodotti Intel dispongono di **avanzate funzionalità di debug**.

Accesso allo **stato interno** delle periferiche nella CPU

⇒ data leak, code execution, security bypass **✗**

Sistema controllo accessi: 

- **Red Unlock: Intel internal** 
- **Orange Unlock: BIOS vendors**
- **Green Unlock: Customers**

Intel Debugging + Exploits!

Ermolov, Sklyarov e Goryachy presentano
Chip Red Pill



Exploit per Intel ME su **Intel Goldmont** (Pentium 6^a gen)

- Red Unlock

Possibilità di **leggere** e **scrivere** microcode

Microcode voltage glitching

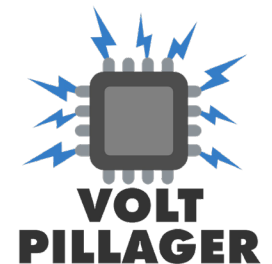
Voltage glitching

- Cosa: generare **glitch**, comportamenti anomali di un obiettivo
- Come: modificando la **tensione** di alimentazione
 - Alterare sistema di alimentazione
- Perché: saltare controlli, leggere dati protetti, eseguire codice...



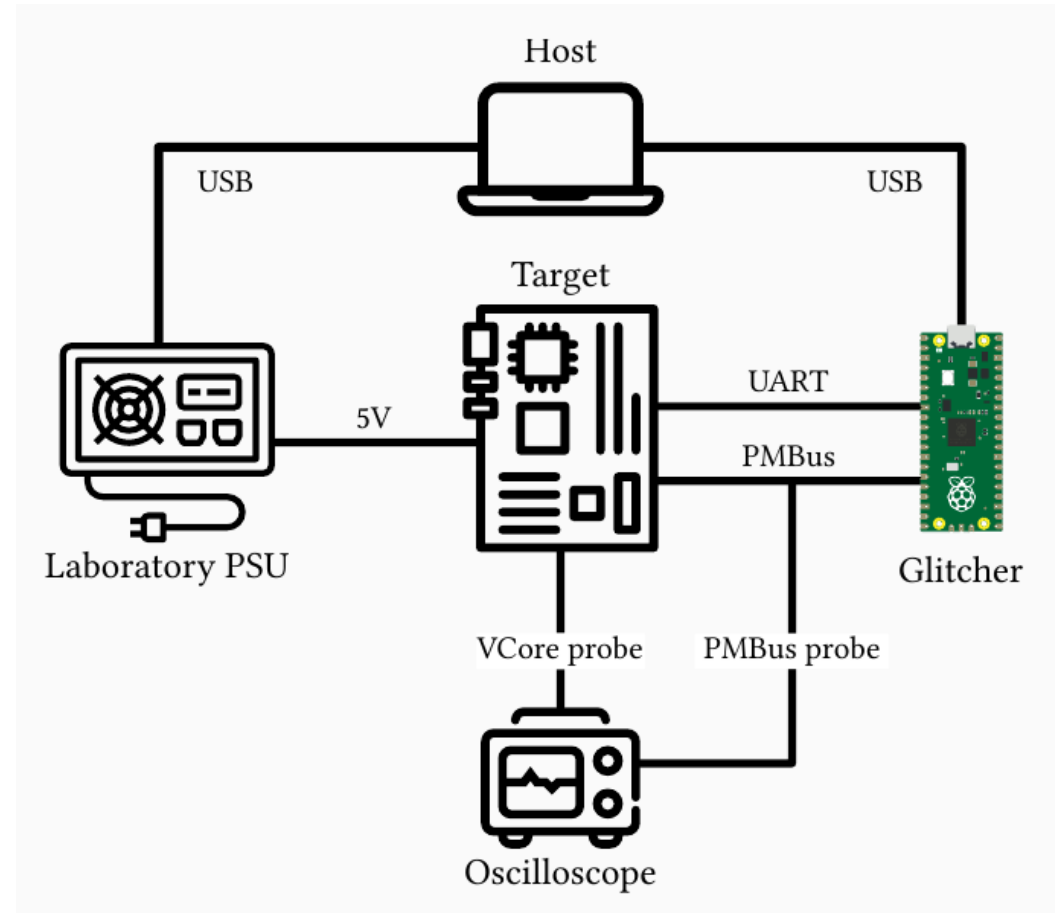
Voltage glitching su x86

- CPU Intel e AMD vulnerabili a **voltage fault injection**
- **Senza modifiche** al sistema di alimentazione
 - Usando il sistema di **undervolting**
- Attacchi a **secure enclave**
- **Microcode?**



Setup hardware

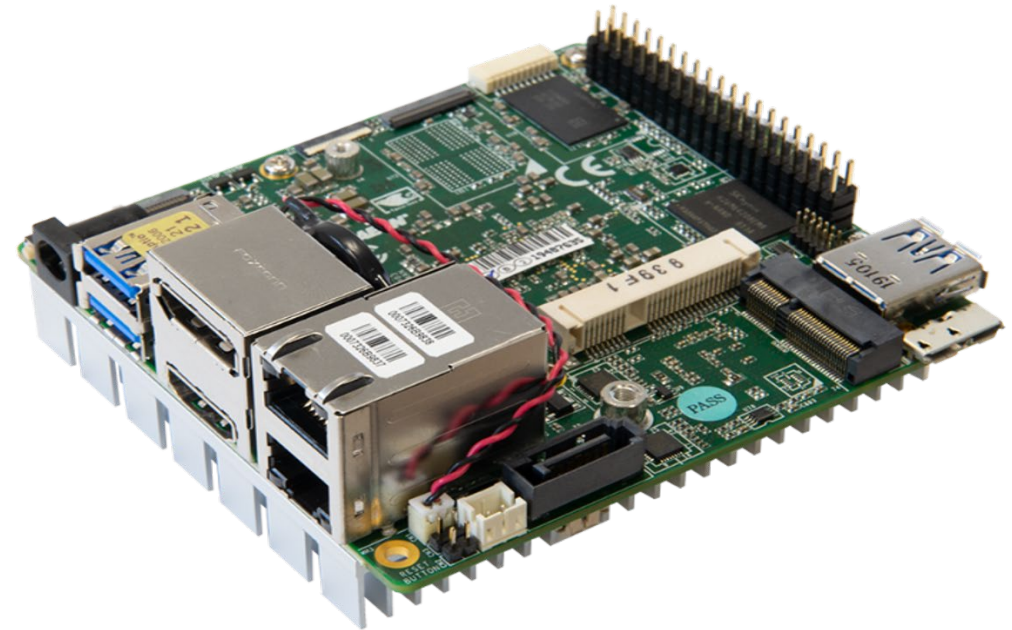
- **Target:** Up Squared board
- **Glitcher:** Raspberry Pi Pico
- Host
- Power supply
- Oscilloscopio



Target Hardware

Up Squared development board

- Intel Pentium N4200
(**Goldmont**)



Target Software

- BIOS con Red Unlock **disponibile!** :)
- Tempo di boot **>2 min** :(
- Non adatto al glitching: migliaia di reset :((

Target Software

- BIOS con Red Unlock **disponibile!** :)
- Tempo di boot **>2 min** :(
 - Non adatto al glitching: migliaia di reset :((
- Up Squared supportata da **coreboot!** :)
 - Tempo di boot **~700 ms**



Glitch roadmap

1. **Replicare** risultati Plundervolt
2. Installare & testare patch **microcode custom**
3. Attaccare microcode **update**

Attacco Assembly IMUL Plundervolt

Risultati Plundervolt:

- Operazioni aritmetiche **non vulnerabili**
- Moltiplicazione IMUL **vulnerabile** con operandi specifici

```
movl $0x80000, %eax;      # operand1
movl $0x4, %ebx;         # operand2
movl %eax, %edx;         # Head
imull %ebx, %edx;        # edx = 0x4*0x80000
movl %eax, %edi;
imull %ebx, %edi;        # edi = 0x4*0x80000
cmp %edx, %edi;
setne %dl;
addb %dl, %cl;           # Tail
```

Listing: IMUL target code

Attacco Assembly IMUL – Miei risultati

- IMUL vulnerabile **indipendentemente** dagli operandi

```
movl $0x80000, %eax;      # operand1
movl $0x4, %ebx;         # operand2
movl %eax, %edx;         # Head
imull %ebx, %edx;        # edx = 0x4*0x80000
movl %eax, %edi;
imull %ebx, %edi;        # edi = 0x4*0x80000
cmp %edx, %edi;
setne %dl;
addb %dl, %cl;           # Tail
```

Listing: IMUL target code

Attacco Assembly IMUL – Miei risultati

- IMUL vulnerabile **indipendentemente** dagli operandi
- È davvero IMUL, oppure **altre operazioni?**

```
movl $0x80000, %eax;      # operand1
movl $0x4, %ebx;         # operand2
movl %eax, %edx;         # Head
imull %ebx, %edx;        # edx = 0x4*0x80000
movl %eax, %edi;
imull %ebx, %edi;        # edi = 0x4*0x80000
cmp %edx, %edi;
setne %dl;
addb %dl, %cl;           # Tail
```

Listing: IMUL target code

Attacco Assembly CMP



🏆 100 - Achievement unlocked
Glitchare CMP nel 75% dei casi

```
movl $0xAAAAAAAA, %eax;  
movl $0xAAAAAAAA, %ebx;  
cmp %eax, %ebx;      # Head  
setne %dl;  
addb %dl, %cl;      # Tail
```

Listing: CMP target code

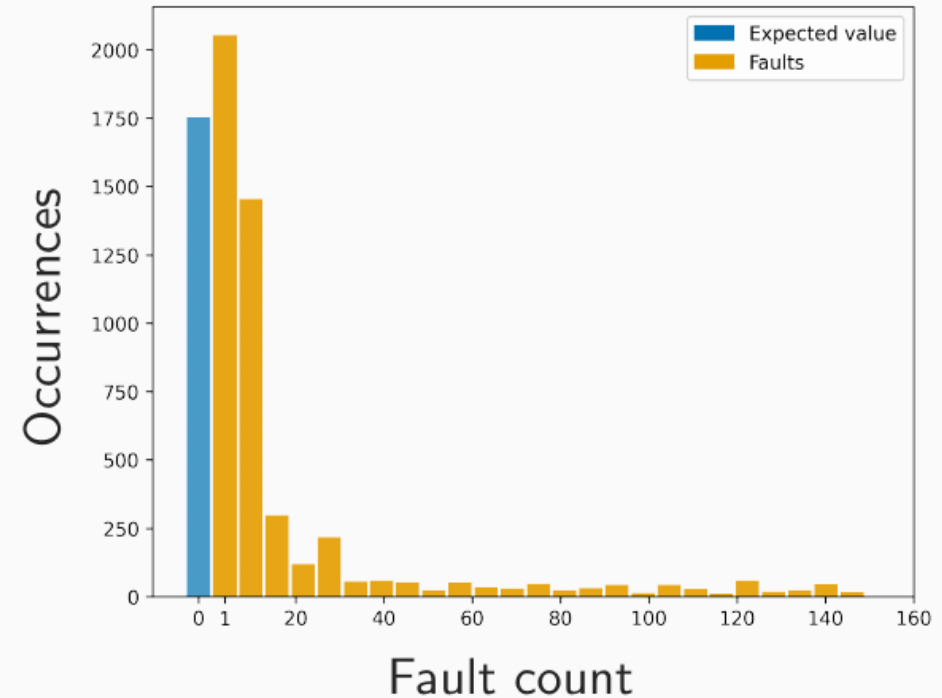


Figure: Fault distribution

Attacco Microcode ADD



🏆 100 - Achievement unlocked
Glitchare il Microcode

```
{  
    ADD_DSZ64_DRI (RCX , RCX , 1) ,  
    ADD_DSZ64_DRI (RCX , RCX , 1) ,  
    ADD_DSZ64_DRI (RCX , RCX , 1) ,  
    NOP_SEQWORD  
},  
/* ... */  
{  
    ADD_DSZ64_DRI (RCX , RCX , 1) ,  
    NOP ,  
    NOP ,  
    END_SEQWORD  
}
```

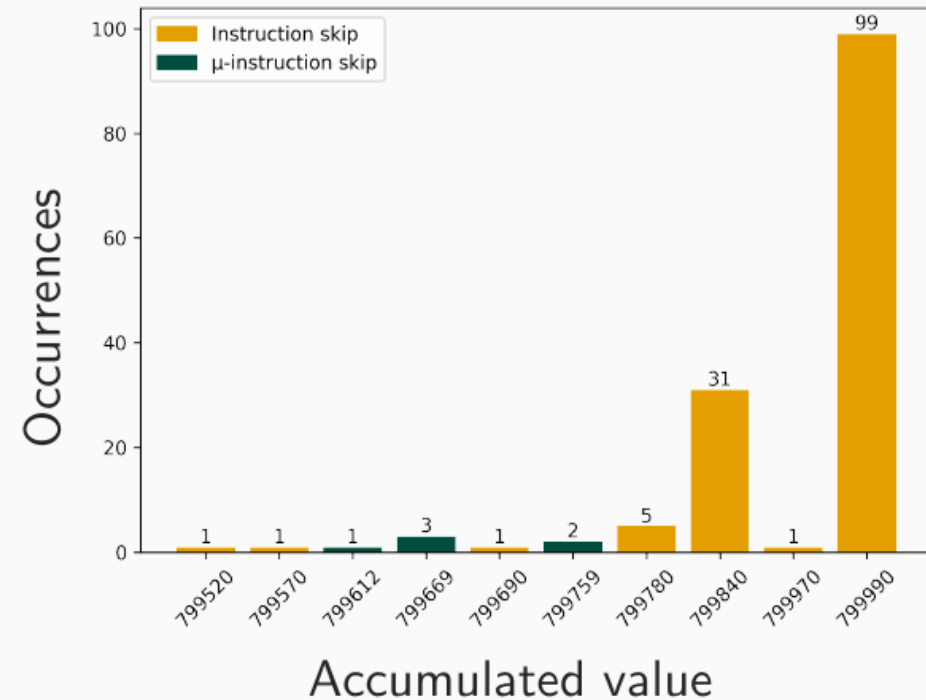
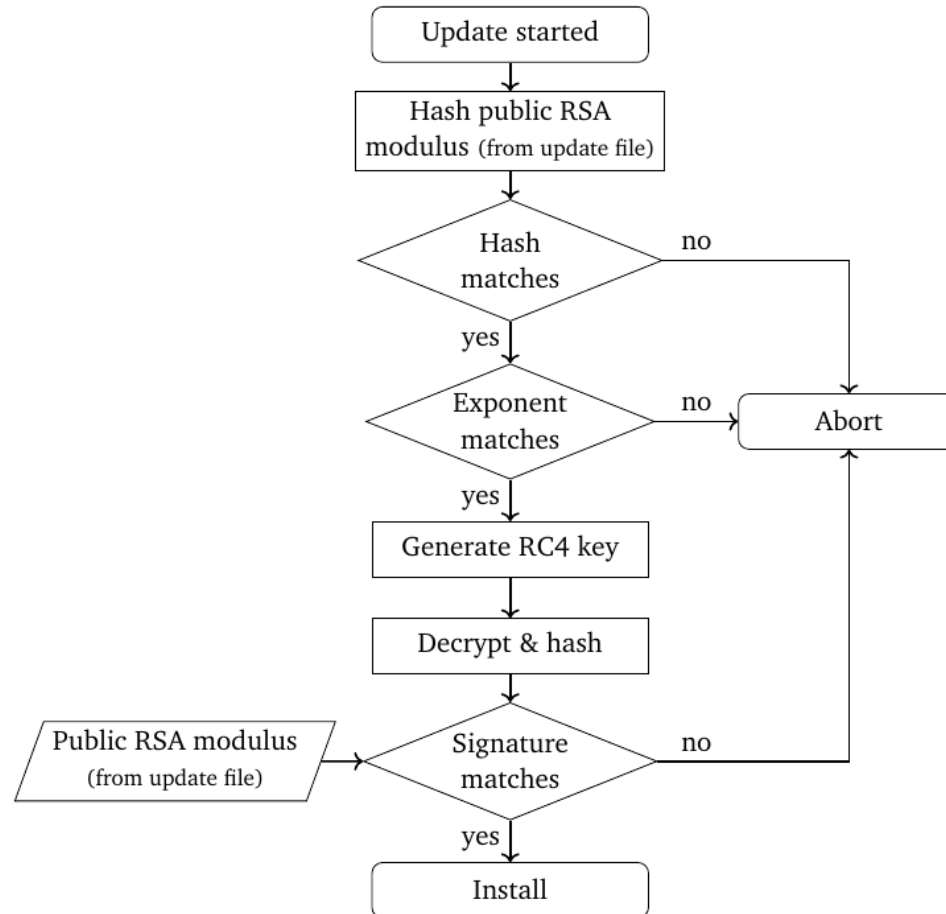


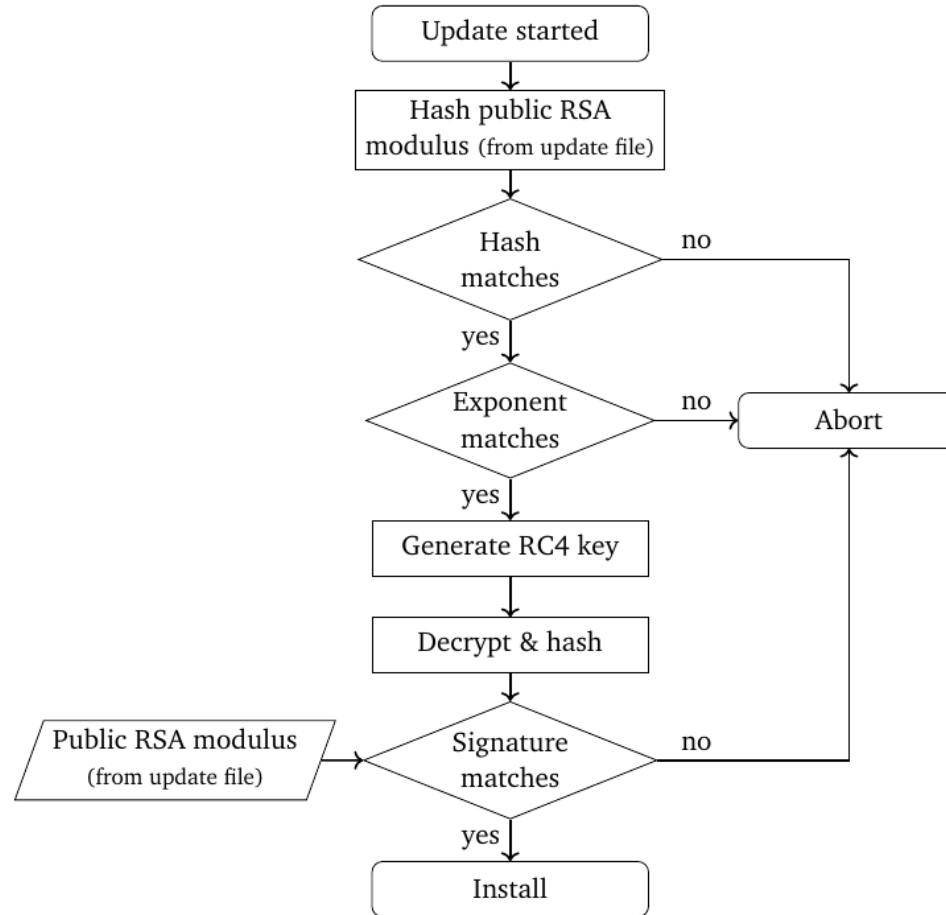
Figure: Skipped (μ)instructions distribution

Secure Microcode Update

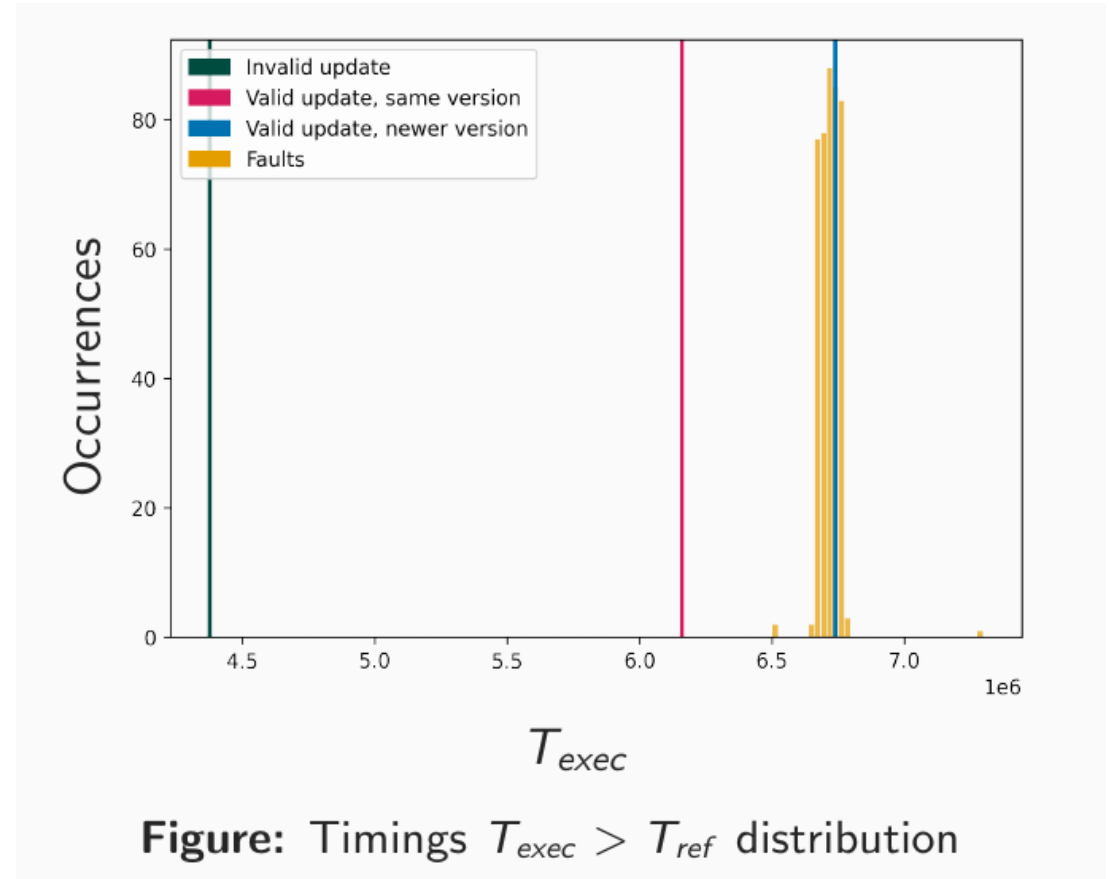
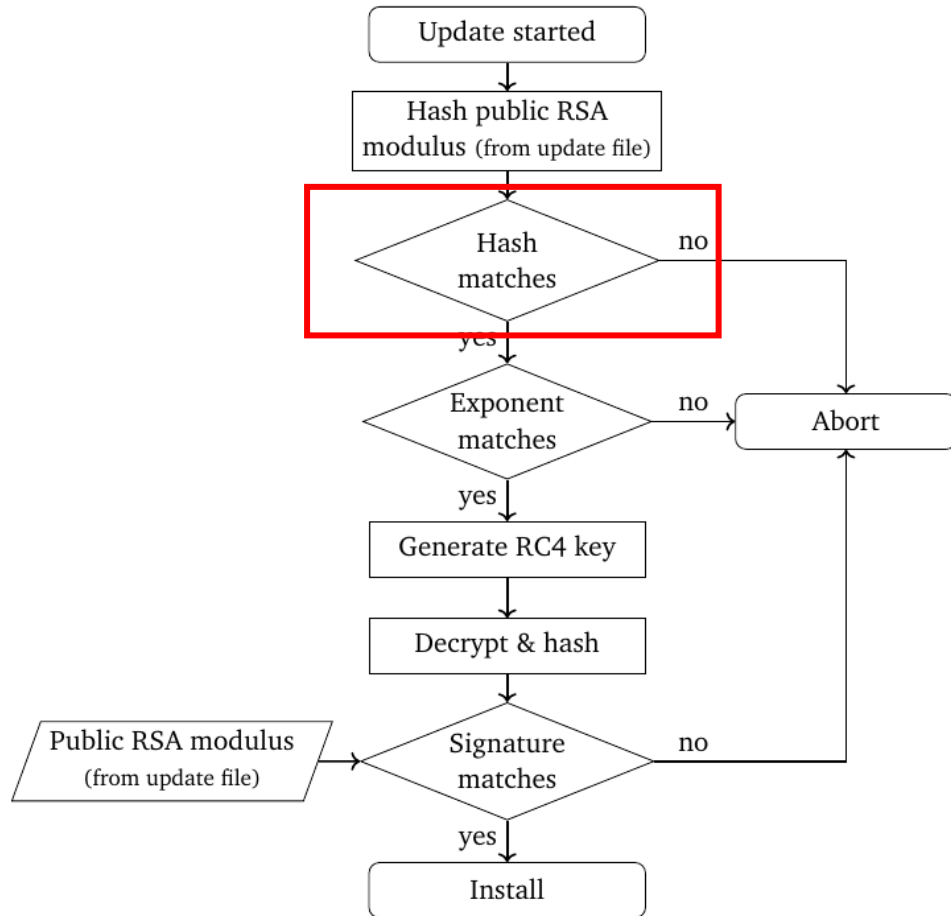


OR IS IT?

Secure Microcode Update



Microcode Update Glitching



Conclusioni

- CISC vs RISC
- Struttura di una CPU CISC
 - Perché microcode?
- Implementazione Intel del microcode
- Debugging su CPU Intel
- Voltage glitching
 - Glitchare operazioni x86
 - Glitchare μ -op
 - Glitchare microcode update

Backup slides

Macroinstruction fusion 1/2

Intel® Core™ Microarchitecture – Front End Intel® Software College

Instruction Decode / Macro-Fusion Absent

Read four instructions from Instruction Queue

Each instruction gets decoded into separate uops

Example



```
for (int i=0; i<100000; i++) {  
    ...  
}
```

Instruction Queue

- add ecx, 1
- mov [mem1], ecx
- mov edx, [mem1]
- cmp eax, [mem2]
- jge label

Cycle 1	add ecx, 1	← dec0
	mov [mem1], ecx	← dec1
	mov edx, [mem1]	← dec2
	cmp eax, [mem2]	← dec3
Cycle 2	jge label	← dec0

Intel® Processor Micro-architecture – Core®

25

Copyright © 2006, Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. Other brands and names are the property of their respective owners.

Macroinstruction fusion 2/2

Intel® Core™ Microarchitecture – Front End Intel® Software College

Instruction Decode / Macro-Fusion Presented

Read five Instructions from Instruction Queue

Send fusable pair to single decoder

Single uop represents two instructions

Example

```
for (unsigned int i=0; i<100000; i++)  
{  
    ...  
}
```



Instruction Queue

- add ecx, 1
- mov [mem1], ecx
- mov edx, [mem1]
- cmp eax, [mem2]
- jae label

Cycle 1

add ecx, 1	← dec0
mov [mem1], ecx	← dec1
mov edx, [mem1]	← dec2
cmpjae eax, [mem2], label	← dec3

Intel® Processor Micro-architecture – Core®

26

Copyright © 2006, Intel Corporation. All rights reserved.
Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. Other brands and names are the property of their respective owners.

Intel CPU vs SoC

