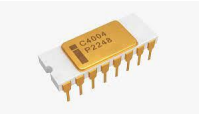
Arkitektur och assembler för INTEL/AMD 64 bitar (x64)



1971

...

https://www.sweclockers.com/test/33062-intel-core-i9-12900k-och-core-i5-12600k-alder-lake

2023

Äldre arkitektur, 16 bitar (x86)

(för bakgrund och historik)

- endast 8 generella(?) register
  - AX för aritmetiska operationer
  - BX för pekare (basadresser)
  - CX för skift och loopar
  - DX för aritmetik och I/O
  - SP stackpekare
  - BP stackbas (framepointer)
  - SI source index för källa vid streaming
  - DI destinationsindex vid streaming

- ordlängd 16 bitar

Arkitektur (x86_64 kallas x64)

- Material på Canvas:
  ”Intel 64-bits arkitektur – Introduktion till assembler-programmering för x86_64”
  (Förutsätter att man kan ARM och grundläggande datorarkitektur.)

- CISC-arkitektur (instruktionsmässigt), med delvis underliggande RISC-arkitektur (dvs hybrid).

- Exekverar instruktioner parallellt => efterföljande synkronisering. Kör pipelining med många exekveringssteg (14-24).

- *Historia: 8086, (80186), 80286, 80386, 80486, pentium XX, Intel Core 2, i7, i3, i5, i9, etc kallas alla för x86. 64-bitarsarkitektur skulle kallas x86_64 men kallas x64.*

- *Intels första processor var 4004 (från 1971) och hade 4-bitarsarkitektur.*

Arkitektur (x64)

Komplicerade CISC-instruktioner (det vi ser)

↓

Omvandlar internt till enkla RISC-instruktioner (under ytan)

↓

Pipeline med 14-24 exekveringssteg, ”out-of-order-exekv.”
Parallellitet i pipeline ger problem:
a) resultat ej färdiga
Lösning:
Stuva om instruktionerna
b) hopp (redan hunnit köra in framtida instr. i pipelinen)
Lösning:
”Deep branch prediction” = gissa framtida hopp och planera pipeline därefter. (Om man gissar rätt = vinst)
”Spekulativ exekvering” = gissa exekveringsväg (inkl hopp). Eventuellt gissas flera olika vägar och alla beräknas.

↓

Slutresultat ”lappas ihop”. ”Nitar kastas”. Skriv till register.

”under ytan – syns ej”

Arkitektur x64

- VARNING: Olika versioner av assembler-översättare:
  - AT&T-syntax (Unix/Linux-världen)
    (gnu-C-kompilator gcc och gnu-assembler gas)
  - Intel-syntax (Windows-världen)

- Ordningen mellan operanderna är omkastad, jämfört med den ARM-assembler vi sett!

- I dokumentet på Canvas (”Intel 64-bits arkitektur – Introduktion till assembler-programmering för x86_64”) samt på labbarna (Lab3) används AT&T-syntax (dvs omkastat mot ARM).

Arkitektur x64

- 16 generella register
  (som jämförelse har 64bits ARM 32 st) (32bits ARM 16 st)

- Ordlängd 64 bitar

- De flesta instruktioner kan jobba mot en operand i register och *en (1)* operand i minnet. (Instruktionerna kan givetvis arbeta med bara registerinnehåll också)
  (jämför ARM: Bara register-register förutom LDR och STR)

Arkitektur x64

- De 16 generella 64-bit registerna kan ”styckas upp i”
  - 32-bitars register
  - 16-bitars register
  - 8-bitars register

Arkitektur (forts.)

Den lägsta halvan av registren (32 bitar) har eget namn

64-bitsregister	32-bitsregister	16-bitsregister	8-bitsregister
rax	eax	ax	al *(ah, hög byte i ax)*
rbx	ebx	bx	bl *(bh, hög byte i bx)*
rcx	ecx	cx	cl *(ch, hög byte i cx)*
rdx	edx	dx	dl *(dh, hög byte i dx)*
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

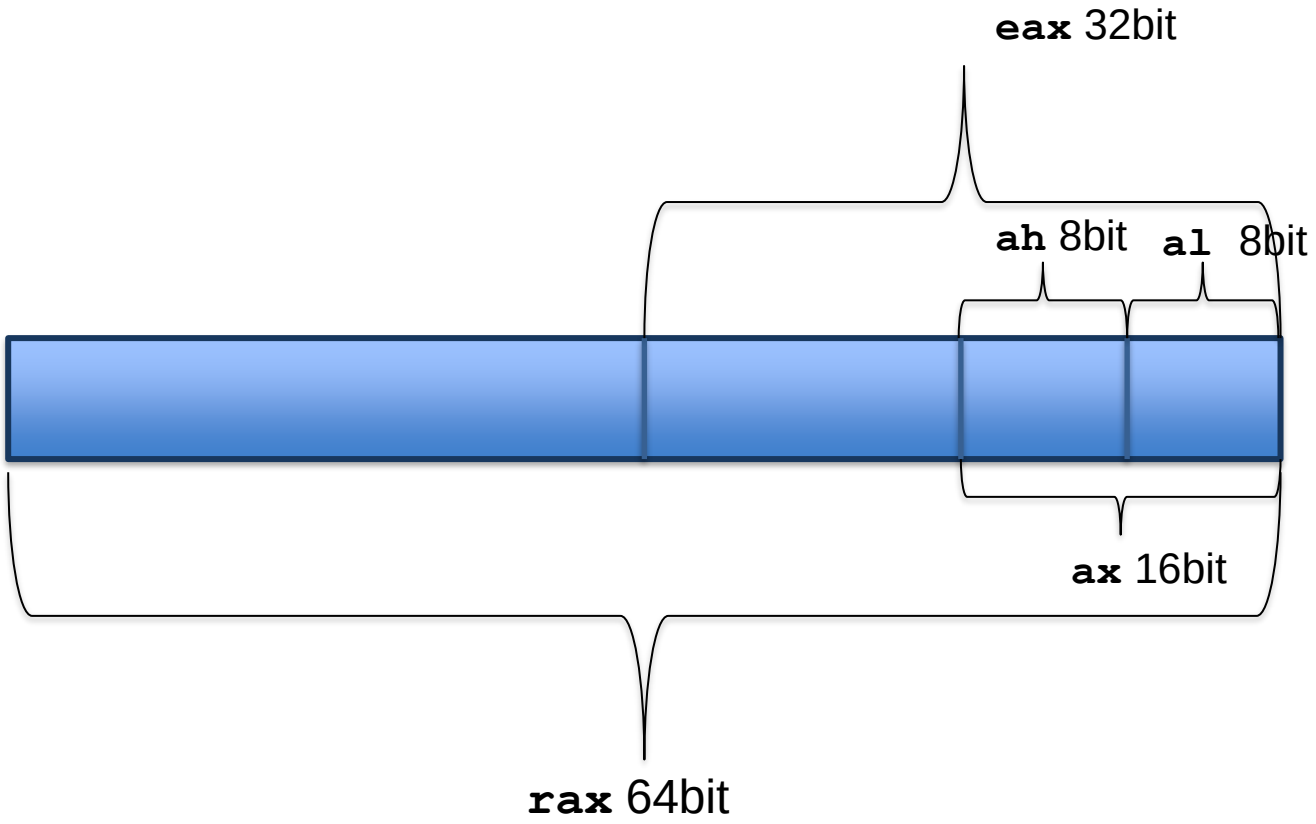
Obs! en rad i tabellen är olika delar av ett och samma register

Samtliga namn (utom specialfallen i parenteser) avser de lägsta bitarna

AX, BX, CX, DX, SP, BP, SI, DI (namn från gamla x86, 16 bit)

- AX för aritmetiska operationer
- BX för pekare (basadresser)
- CX för skift och loopar
- DX för aritmetik och I/O
- SP stackpekare
- BP stackbas (framepointer)
- SI source index för källa vid streaming
- DI destinationsindex vid streaming

64-bits register



Suffix till instruktioner

- Suffix till instruktioner anger hur stort dataformat som ska användas
  - **b = byte (8 bitar)**
  - *s = short (16-bits heltal) eller single (32-bits flyttal)*
  - **w = word (16 bitar,** OBS!!)
  - **l = long (32-bits heltal** eller 64-bits flyttal)
  - **q = quad (64 bitar)**
  - *t = ten bytes (80 bits flyttal)*
- Om man utelämnar suffix används formatet hos destinationsregistret (osäker programmering, **rekommenderas inte**)

Några exempel

OBS! Ordningen omvänd mot ARM

Notera att ena inparametern skrivs över.

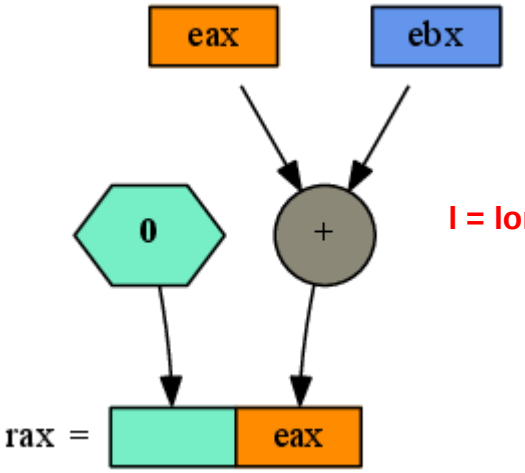
q = quad = 64 bitar

64 bitar 64 bitar

$ anger immediate data (jfr # i ARM)

% anger register

```
addq %rbx,%rax      # rax <- rax+rbx
addq $-1,%rax       # rax <- rax-1
decq %rax           # rax <- rax-1
```



- Instruktioner som skriver över lägre halvan av ett register lägger nollor i övre halvan:

l = long = 32 bitar 32 bitar 32 bitar

```
addl %ebx,%eax      #eax <- eax+ebx
```

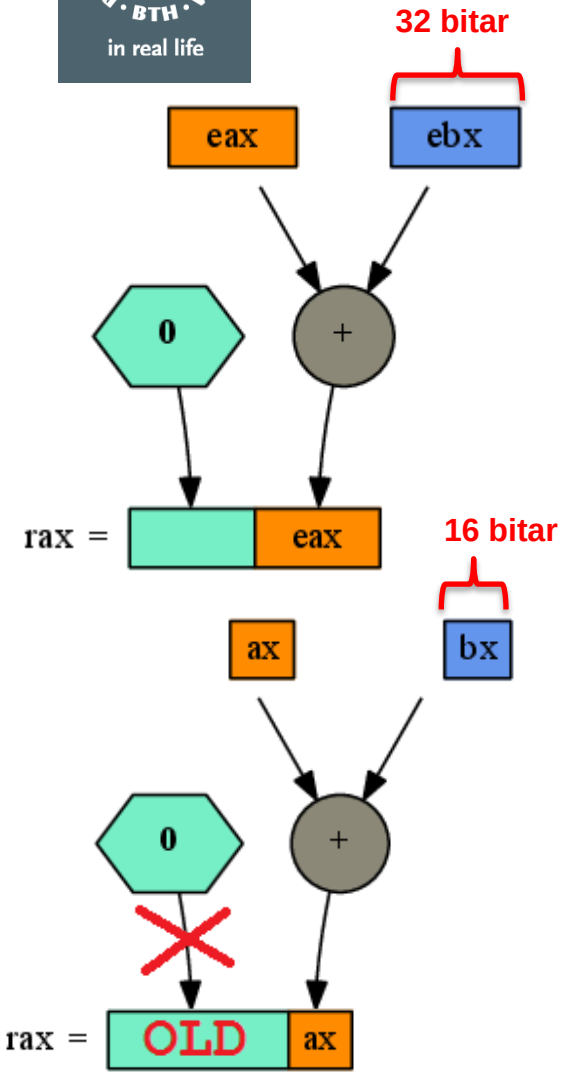
64 bitar 32 bitar

```
add %rbx,%eax       #fungerar inte!
```

(Formatet måste vara lika stort!)

Inget suffix => eax
(destinationsregister) bestämmer (32 bitar)
(osäker programmering, rekommenderas inte)

OBS!!!



- Att den högre delen av registret fylls med nollor gäller ***bara instruktioner som skriver 32 bitar***.

- Om man skriver 16 eller 8 bitar i ett register kommer resten att vara oförändrat. (Nollställ själv!)

Notera att rax, eax, ax, ah och al är samma register

Indirekt adressering till minne

Parentes anger att registrets innehåll tolkas som adress
(Jfr [ ] för ARM)

64 bitar 32 bitar

l = long = 32 bitar

```
movl  (%rbx),%eax     #laddar ett 32-bitstal från
                      #minnesadressen rbx pekar på till eax
```

(%rbx) = 64-bitars adress som pekar på ett 32-bitars tal

64 bitar 64 bitar

q = quad = 64 bitar

```
movq  %rdi,(%r12)     #sparar 64 bitar från rdi till
                      #den minnesplats r12 pekar på
```

32 bitar 64 bitar

l = long = 32 bitar

```
movl  %eax,-4(%rbp)   #sparar 32 bitar från eax till
                      #adressen rbp-4
```

Minnesadressen i register rbp ökas med (-4)

Jfr [r1,#4] i ARM

Alternativ adressering till minne

Istället för t.ex. **`(%rbx)`** kan man använda:

- Deklarera en variabel (t.ex. **`SUM:`**) i programmets datasektion. Då kan **`SUM`** användas direkt i instruktionen.

Anm: Med **`lea`** *(load effective address)* flyttar man en adress till ett register.

Exempel **`leaq SUM,%r11`**

q = quad = 64 bitar

- Ange ett tal direkt i instruktionen med hjälp av $

Detta kallas *immediate adressering*.

Exempel: $4 (Notera att det tolkas som 32 bitar.)

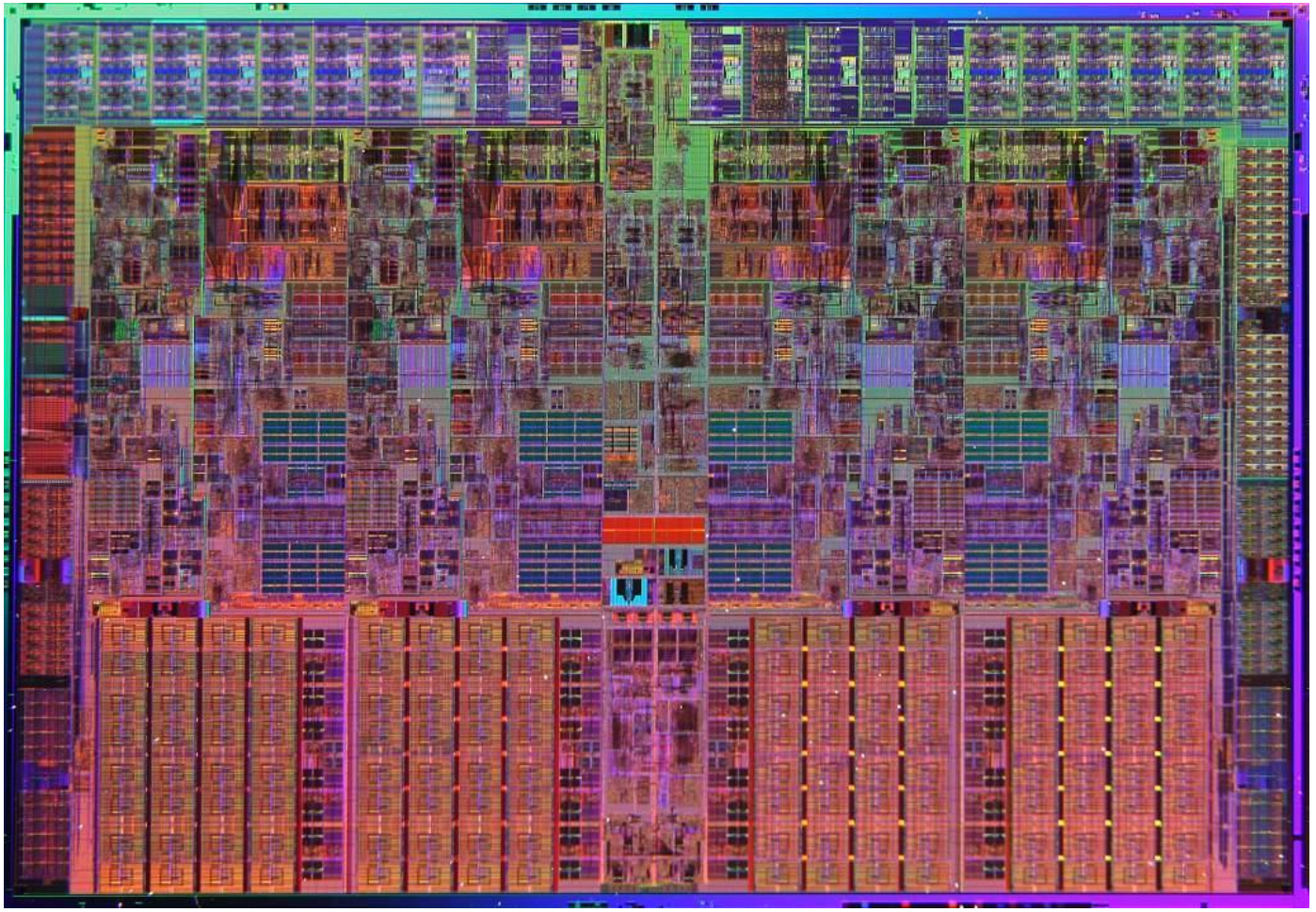
Hantering av stacken

64 bitar

- Registret **`rsp`** används normalt som stackpekare
- OBS! Stacken växer mot lägre adresser
- Instruktionerna **`push`** och **`pop`** sparar respektive hämtar data på stacken och uppdaterar stackpekaren automatiskt
- instruktionen **`call`** (som används för hopp till subrutin) ”pushar” automatiskt återhoppsadressen på stacken.
- instruktionen **`ret`** används vid återhopp från subrutin och ”popar” automatiskt återhoppsadressen från stacken till programräknaren

(Vi slipper problemet som i ARM med att länkregistret skrivs över vid subrutin i subrutin.)

Vad döljer sig i ”Intel Inside”?



Moderna processorer

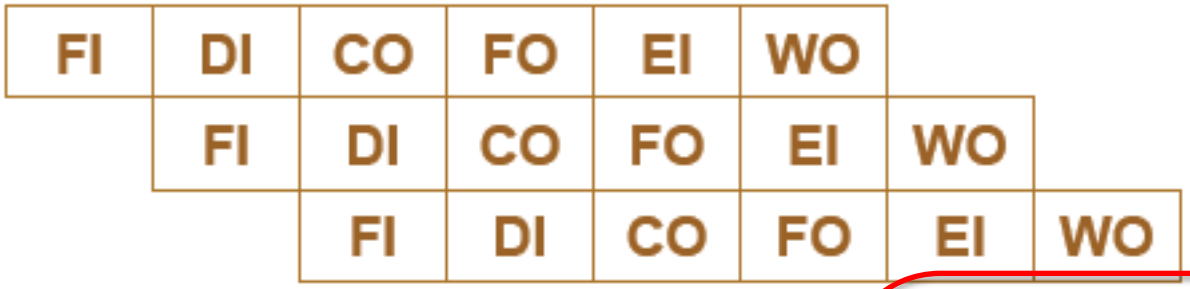
Bygger på *superskalära* pipelinade strukturer med spekulativa metoder för ”*out-of-order*”- exekvering

- Superskalär: Kan exekvera mer än en skalär *(heltals-)* instruktion åt gången

- Out-of-order: Kan exekvera instruktioner i en annan ordning än de står i programmet, blockerar ej varandra

Principiell pipeline

- Förenklad till 6 steg
  (modern Intel har 14 (Penryn) – 24 (Nehalem))



Jfr ARM

- Fetch instruction (FI) (jfr IF = Instruction Fetch)
- Decode instruction (DI) (jfr ID = Instruction Decode)
- Calculate operand address (CO)
- Fetch operand (FO)
- Execute instruction (EI) (jfr EXE = Execute)
  (jfr MEM = memory access)
- Write operand (WO) (jfr WB = Write Back)

Instruction fetch unit och instruktionskö

- Det finns en *fetch unit* som hämtar instruktioner ***innan*** de behövs

- Dessa instruktioner lagras i en instruktions-kö

- *Fetch unit* kan känna igen hoppinstruktioner och generera hoppadress => kostnad för ovillkorliga hopp minskar. *Fetch unit* kan alltså hämta instruktioner enligt hopp.

- För villkorliga hopp är det svårare, då måste man veta om hoppet ska tas eller inte

Hantering av hopp (branches)

- Stall
- Delayed branching
- Branch prediction:
  - Statisk prediktering
  - Dynamisk prediktering
  - Spekulativ branch prediktion

Delayed branching
(repetition från pipelining)

- Om kompilatorn inte hittar en lämplig instruktion att lägga i delay slots, så läggs en **`NOP`** (no operation) in.

- I ett normalt program kan kompilatorn i ca 60 – 85% av fallen med hoppinstruktioner hitta en annan lämplig instruktion flytta till *branch delay slot*

Statisk *branch prediction*

- Vid **statisk branch prediction** tas ingen hänsyn till exekveringshistoriken

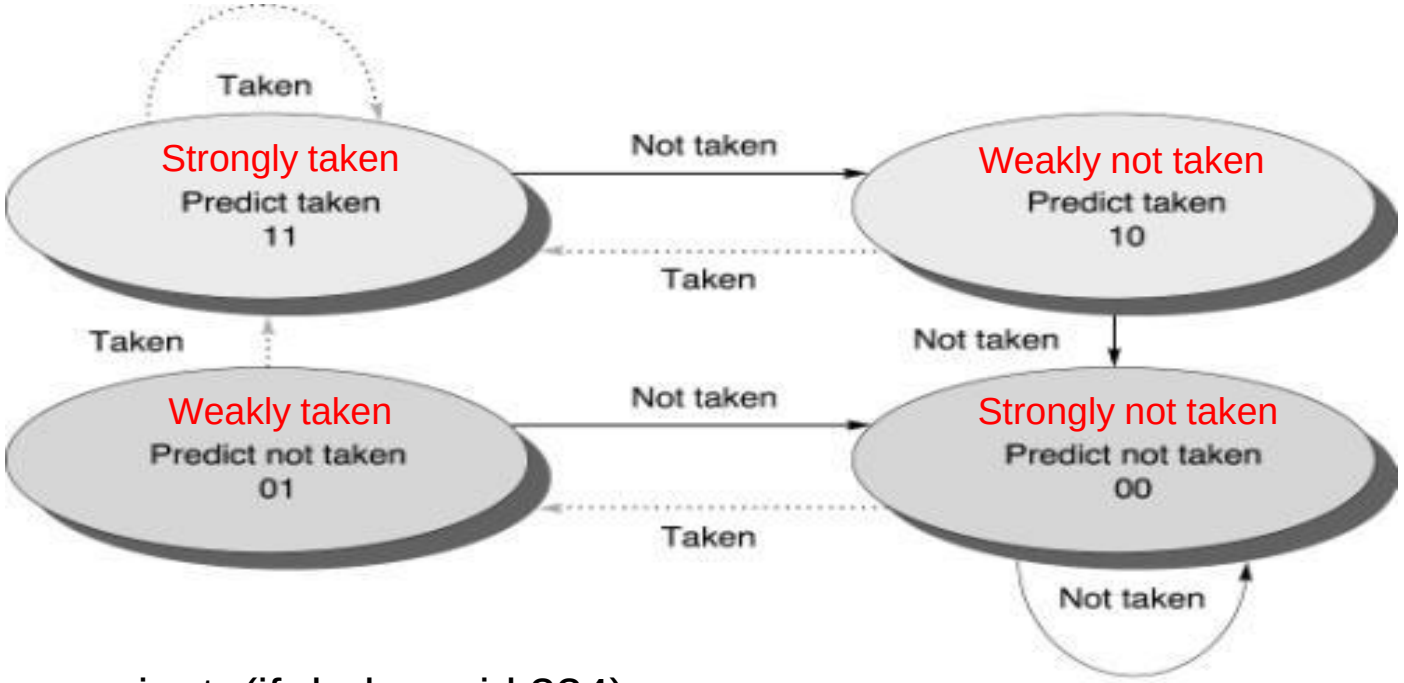
- Olika statiska principer
  - Predict never taken – antar att hoppet **aldrig** utförs.
  - Predict always taken – antar att hoppet **alltid** utförs.
  - Prediktion beroende på riktning
    - Predict branch taken – för tillbakahopp
    - Predict branch not taken – för hopp framåt

Dynamisk *branch prediction*

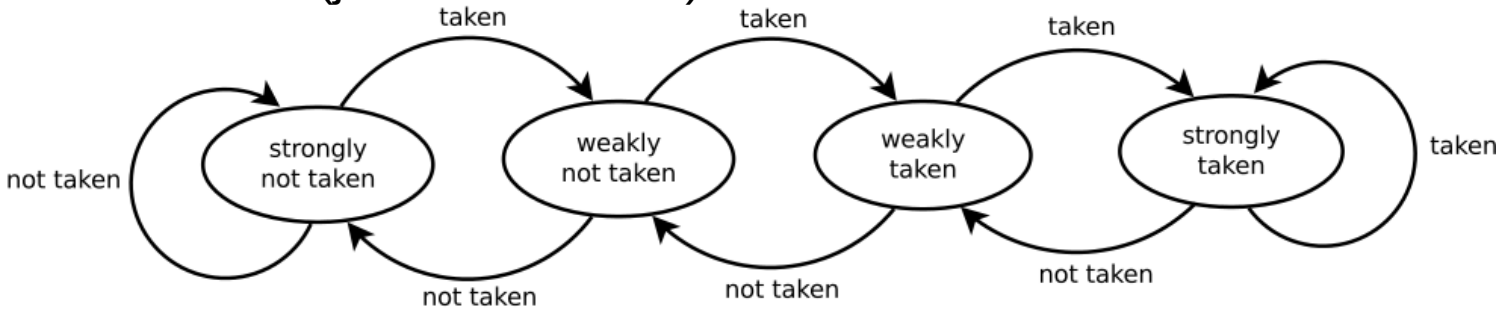
- I dynamisk prediktering tas hänsyn till exekveringshistoriken
- En bit för prediktion
  - Sparar ifall hoppet togs förra gången hoppinstruktionen användes och predikterar (gissar) att samma sak ska hända som förra gången. Om hoppet inte togs förra gången gissar man det inte ska tas nu heller och vice versa.
- Två bitar för prediktion
  - En mer ”kvalificerad gissning”

    Ex: ”En prediktion skall vara fel två gånger för att den skall ändras”
- Branch prediction buffer (*branch history table*)
  - Ett litet minne indexerat med lägsta adressbitarna
  - Innehåller prediktionsbitarna om hopp från aktuell instruktionsadress

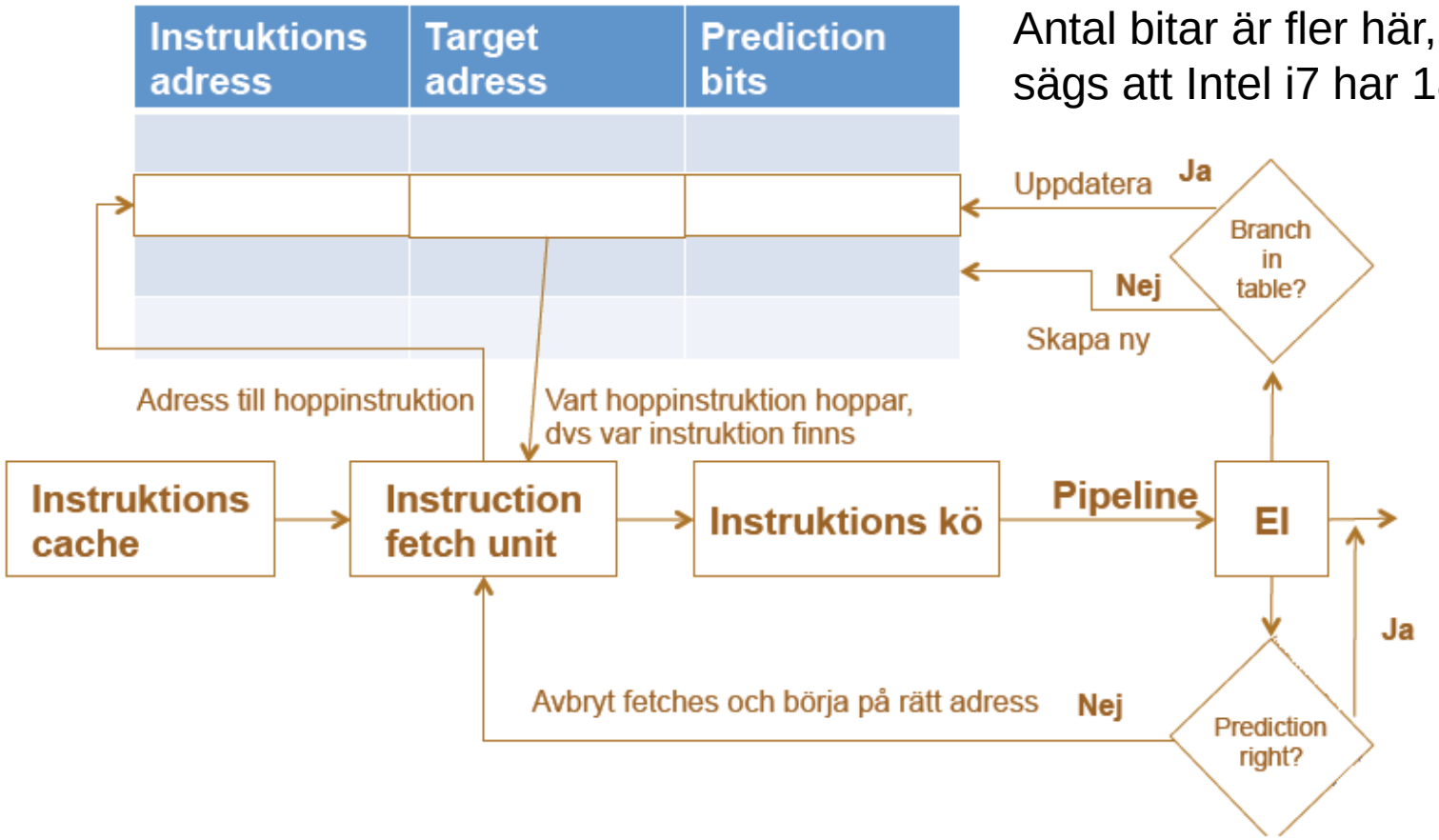
Dynamisk branch prediction, 2 bitar



Annan variant: (jfr boken sid 334):



Branch target buffer
(prediktionscache med hoppadresser)



Antal bitar är fler här,
sägs att Intel i7 har 18 bitar

EI = Execute Instruction

Branch prediction – villkorliga hopp

Beskrivning av programmet:
- `%rcx <- %rcx+%rbx`
- Hoppa till LABEL om noll
- Multiplikation: `R[%rdx]:R[%rax]←S·R[%rax]`
- Flytta talet 10 till register `%rsi`

S=Source, dvs %rax

Antagande (prediction):
Nästa instruktion exekveras (inget hopp)

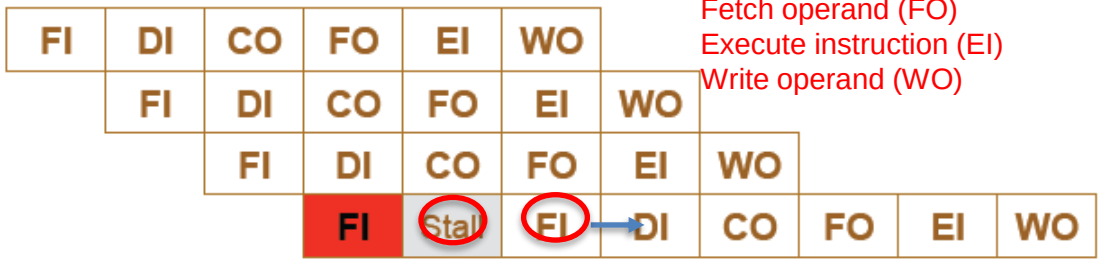
- Alternativ 1: Hoppet görs inte (antagandet var rätt)

```
addq    %rbx,%rcx
je      LABEL
mulq    %rax
movq    $10,%rsi
```

- Kostnad 1 cykel

- Alternativ 2: Hoppet görs (antagandet var fel)

```
addq    %rbx,%rcx
je      LABEL
mulq    %rax
instr vid LABEL
```



Fetch instruction (FI)
Decode instruction (DI)
Calculate operand address (CO)
Fetch operand (FO)
Execute instruction (EI)
Write operand (WO)

- Kostnad 2 cykler

Kommentar: `mulq` *S* betyder **R**[%rdx]: **R**[%rax] ← *S* × **R**[%rax] (dvs `mulq %rax` blir kvadraten)

Branch prediction forts.

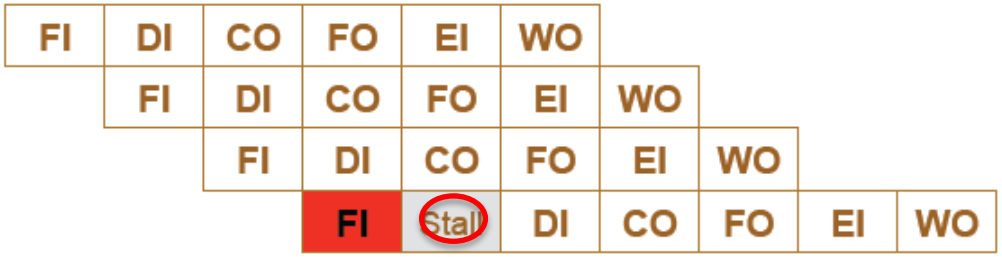
Beskrivning av programmet:
- `%rcx <- %rcx+%rbx`
- Hoppa till LABEL om noll
- Multiplikation: `R[%rdx]:R[%rax]←S·R[%rax]`
- Flytta talet 10 till register `%rsi`

S=Source, dvs %rax

Antagande (prediction):
Instruktion vid LABEL exekveras (hoppet görs)

- Alternativ 1: Hoppet görs (antagandet var rätt)

```
addq    %rbx,%rcx
je      LABEL
mulq    %rax
instr vid LABEL
```

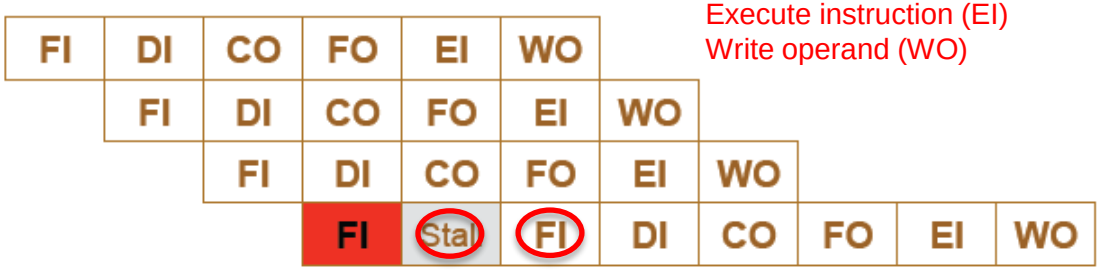


- Kostnad 1 cykel

Fetch instruction (FI)
Decode instruction (DI)
Calculate operand address (CO)
Fetch operand (FO)
Execute instruction (EI)
Write operand (WO)

- Alternativ 2: Hoppet görs inte (antagandet var fel)

```
addq    %rbx,%rcx
je      LABEL
mulq    %rax
movq    $10,%rsi
```



- Kostnad 2 cykler

Kommentar: `mulq S` betyder **R**[%rdx]: **R**[%rax] ← *S* × **R**[%rax] (dvs `mulq %rax` blir kvadraten)

Branch prediktion forts.

- Rätt ***branch prediction*** är viktigt *(intelligent spågumma)*
- Baserat på prediktion kan en instruktion och de som förmodas följa efter den hämtas och placeras i instruktionskön
- När hoppvillkoret är bestämt kan exekveringen fortsätta
- Om gissningen är fel måste ”rätt” instruktioner hämtas
- För att utnyttja *branch prediction* maximalt kan exekveringen påbörjas innan hoppvillkoret är bestämt – kallas **spekulativ exekvering**

Spekulativ exekvering

- Med spekulativ exekvering menas att delar av instruktioner exekveras innan processorn vet om det är rätt instruktioner som ska exekveras.

- Om gissningen var rätt kan processorn fortsätta, annars får den göra om (hämta rätt instruktion)

Spekulativ exekvering forts.

Beskrivning av programmet:
- `%rcx <- %rcx+%rbx`
- Hoppa till LABEL om noll
- Multiplikation: `R[%rdx]:R[%rax]←S·R[%rax]`
- Flytta talet 10 till register `%rsi`

S=Source, dvs %rax

Antagande (prediction):
Instruktion vid LABEL exekveras (hoppet görs)

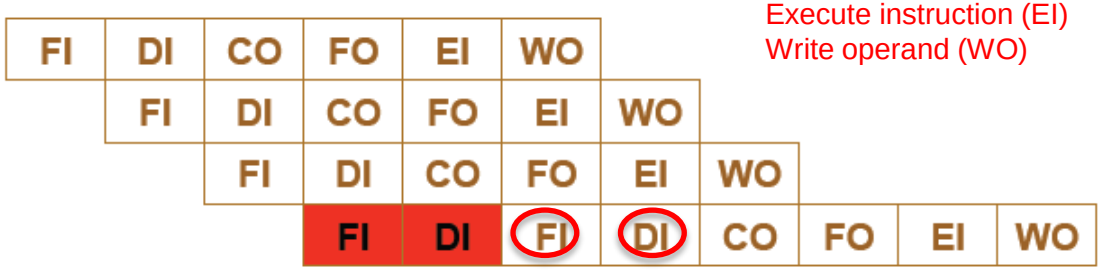
- Alternativ 1: Hoppet görs (antagandet var rätt)

```
addq    %rbx,%rcx
je      LABEL
mulq    %rax
instr vid LABEL
```

- Kostnad 0 cykler

- Alternativ 2: Hoppet görs inte (antagandet var fel)

```
addq    %rbx,%rcx
je      LABEL
mulq    %rax
movq    $10,%rsi
```



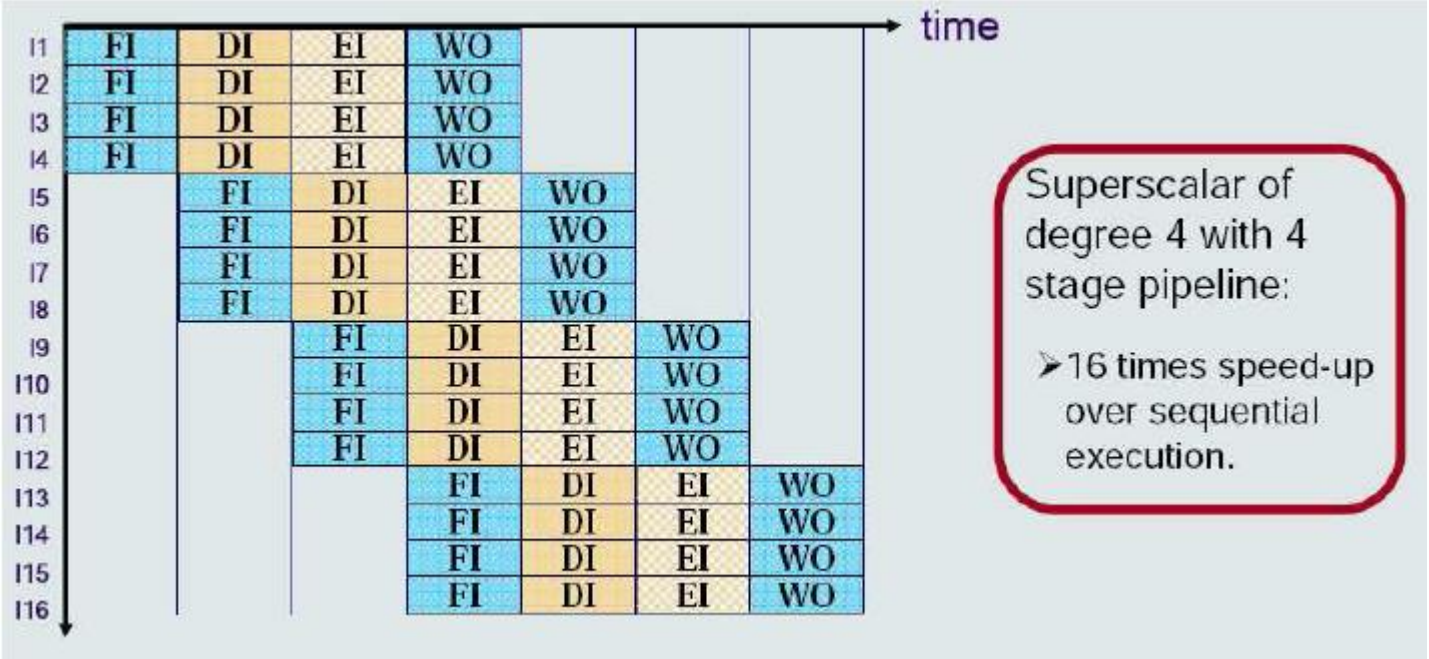
- Kostnad 2 cykler

Fetch instruction (FI)
Decode instruction (DI)
Calculate operand address (CO)
Fetch operand (FO)
Execute instruction (EI)
Write operand (WO)

Kommentar: `mulq S` betyder **R**[%rdx]: **R**[%rax] ← *S* × **R**[%rax] (dvs `mulq %rax` blir kvadraten)

Superskalär arkitektur

- Kan exekvera mer än en instruktion åt gången eftersom de har mer än en pipeline



Exempel

- Intel core i7 har t.ex. 4 st parallella pipelines.

Out-of-order exekvering

- Hitta instruktioner oberoende av varandra och försöker exekvera dem parallellt
- Det innebär att exekveringsordningen kan förändras gentemot ursprungsprogrammet
- Programmets resultat får dock inte bli annorlunda än om instruktionerna körts i sekvens

Out-of-order exekvering

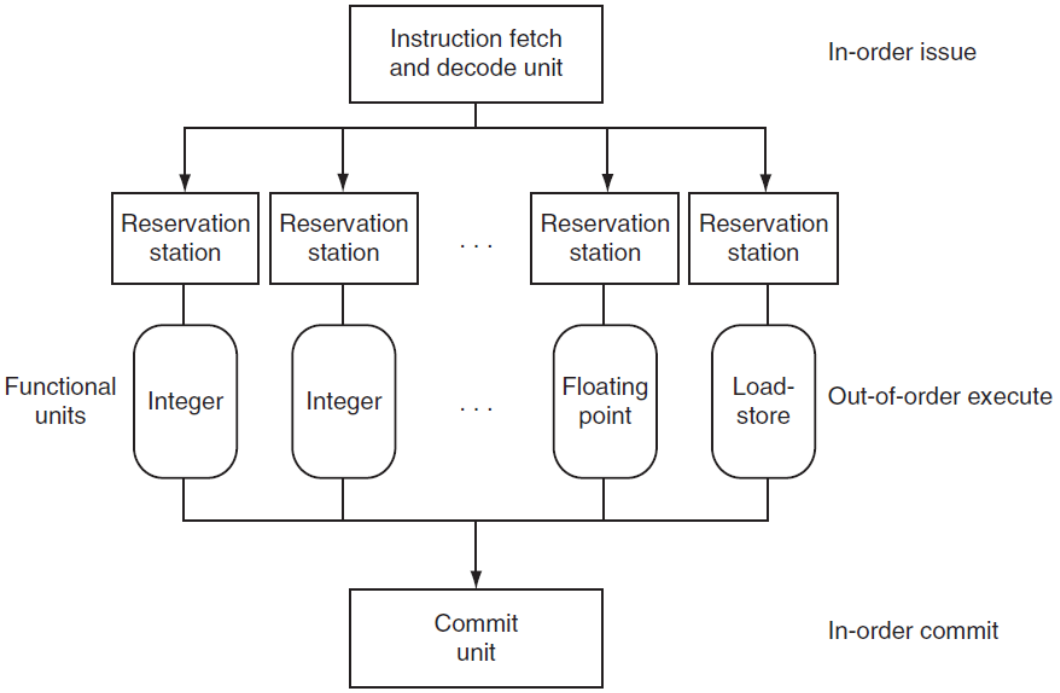


FIGURE 4.70 The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation.

Kapacitetsutnyttjande

- Utnyttjandegraden är ofta låg, beroende på
  - Resurskonflikter
  - Databeroenden
  - Villkorliga instruktioner och hopp

- Ett sätt att utnyttja exekveringskapaciteten bättre är så kallad ”hyperthreading”.
  - Två trådar körs in i strukturen för en kärna för att kunna fylla pipelinerna bättre (fler oberoende instruktioner att välja på). Normalt ”tror” operativsystemet att det är två kärnor istället för en fysisk. Operativsystemet måste kunna hantera flera kärnor.

- Ett sätt att minska databeroenden är så kallad *register renaming (register aliasing)*, vilket innebär att man använder mer än ett fysiskt register till samma variabel för att eliminera ”falska” databeroenden.

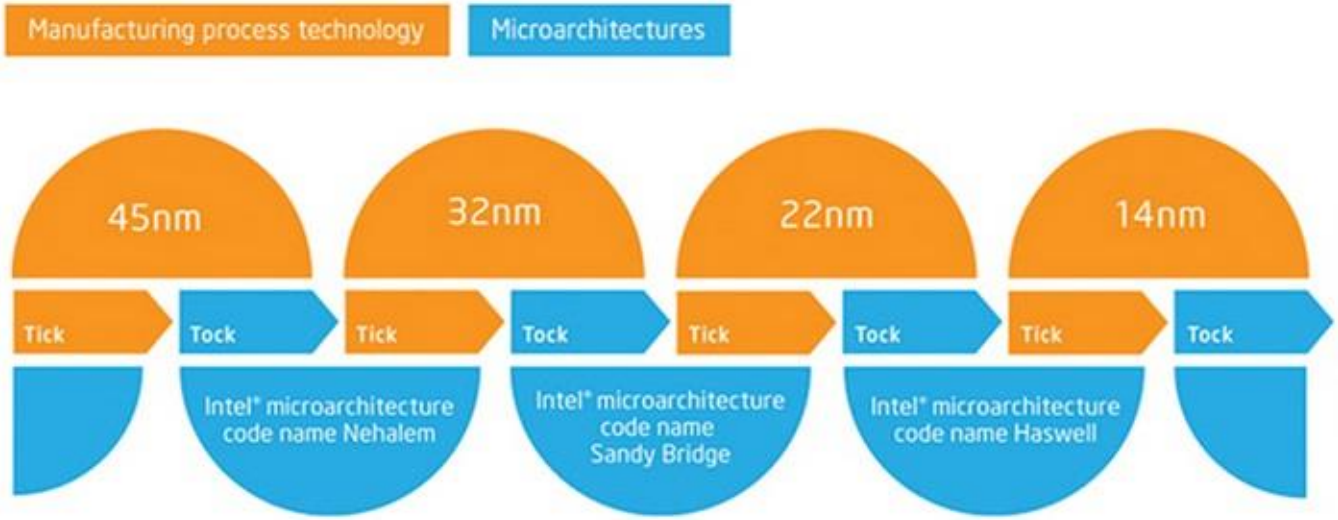
Utvecklingssteg enligt ”tick-tock”modell

- ”tick” krympning av halvledarprocessen, samma µarkitektur
- ”tock” ny µarkitektur, samma tillverkningsprocess
- nytt steg varje år planerat (sackar efter något)

Architectural change		Codename	uArch	Process	Release date
Tick	New Process			65 nm	Jan 5, 2006
Tock	New uArch	Conroe	Core		July 27, 2006
Tick	New Process	Penryn		45 nm	Nov 11, 2007
Tock	New uArch	Nehalem	Nehalem		Nov 17, 2008
Tick	New Process	Westmere		32 nm	Jan 4, 2010
Tock	New uArch	Sandy Bridge	Sandy Bridge		Jan 9, 2011
Tick	New Process	Ivy Bridge		22 nm	2012
Tock	New uArch	Haswell	Haswell		2013
Tick	New Process	Broadwell		14 nm	2014
Tock	New uArch	Skylake i9	Skylake i9		2015
Tick	New Process	~~Skymont~~ Cannon Lake		10 nm	~~2016~~ 2017
Tock	New uArch	Ice Lake (Sunny Cove)	Ice Lake (Sunny Cove)		~~2017~~ 2018
Tick	New Process				
Tock	New uArch				

Anm: nm = nanometer = linjebredd -> påverkar tätheten (ledningslängder, dvs fördröjningar, dvs snabbhet) samt antal komponenter (dvs antal processorkärnor.)

Tic-Tac-Toe



	Tick	Tock	Toe
45nm	Penryn	Nehalem	-
32nm	Westmere	Sandy Bridge	-
22nm	Ivy Bridge	Haswell	Devil's Canyon
14nm	Broadwell	Sky Lake	Kaby Lake
10nm	Cannon Lake (2017)	Ice Lake (2018)	Tiger Lake (2019)

Source: Wikipedia

- På sistone verkar det som om Intel arbetar i tre steg istället
  (toe-steget avser optimering av redan befintlig arkitektur och tillverkningsprocess)

Lite mer detaljerat:

https://www.theverge.com/2020/9/2/21408718/intel-11th-gen-tiger-lake-cpu-processor-announcement-laptops-fall

Tiger Lake i3-i7 för bärbara (lågt pris)

Fabrication process	Micro-architecture	Code names	Core i generation	Release date	Desktop	Mobile	Enthusiast/ WS	2P/4P Server/WS	4P/8P Server
					Processors				
45 nm	Nehalem	Nehalem	Previous[10]	2008-11-17 [11]	Lynnfield	Clarksfield	Bloomfield	Gainestown	Beckton
32 nm		Westmere		2010-01-04 [12][13]	Clarkdale	Arrandale	Gulftown	Westmere-EP	Westmere-EX
32 nm	Sandy Bridge	Sandy Bridge	2	2011-01-09 [14]	Sandy Bridge	Sandy Bridge-M	Sandy Bridge-E	Sandy Bridge-EP	–[15]
22 nm		Ivy Bridge	3	2012-04-29	Ivy Bridge	Ivy Bridge-M	Ivy Bridge-E [16]	Ivy Bridge-EP [17]	Ivy Bridge-EX [17]
22 nm	Haswell	Haswell	4	2013-06-02	Haswell-DT [18]	Haswell-MB (37–57W TDP, PGA package) Haswell-H (47W TDP, BGA package) Haswell-ULP/ULX (11.5–15W TDP)[18]	Haswell-E	Haswell-EP	Haswell-EX
		Devil's Canyon		2014-06	Haswell-DT	N/A			
14 nm		Broadwell	5	2014-09-05	Broadwell-DT	Broadwell-H (37–47W TDP) Broadwell-U (15–28W TDP) Broadwell-Y (4.5W TDP)	Broadwell-E	Broadwell-EP [19]	Broadwell-EX [19]
14 nm	Skylake	Skylake	6	2015-08-05 [20]	Skylake-S	Skylake-H (35–45W TDP) Skylake-U (15–28W TDP) Skylake-Y (4.5W TDP)	Skylake-X [21] Skylake-W	Skylake-SP (formerly Skylake-EP/-EX)[22]	
14 nm		Kaby Lake	7	2016-10	Kaby Lake-S	Kaby Lake-H (35–45W TDP) Kaby Lake-U (15–28W TDP) Kaby Lake-Y (4.5W TDP)	Kaby Lake-X [21]		
14 nm		Kaby Lake Refresh	8	2017-09	N/A	Kaby Lake-U (15W TDP)	N/A		
14 nm		Coffee Lake	8 / 9	2017-10 [23]	Coffee Lake-S	Coffee Lake-B ? Coffee Lake-H Coffee Lake-U	Coffee-Lake-S ?		
14 nm		Kaby Lake G	8	2018-01-07 [24]	N/A	Kaby Lake-G ?	N/A		
10 nm		Cannon Lake		2018-05		Cannon Lake-U			
14 nm		Whiskey Lake		2018-08-28		Whiskey Lake-U			
14 nm		Amber Lake				Amber Lake-Y			
14 nm		Cascade Lake	N/A	2019-04-02			Cascade Lake-X	Cascade Lake-SP	
14 nm		Cooper Lake		2019			Cooper Lake-X	Cooper Lake-SP	
14 nm		Comet Lake	9	2019 / 2020	Comet Lake-S	Comet Lake-U[25][26][27][28]			
10 nm	Ice Lake[29] (Sunny Cove[30])	Ice Lake[31] (Sunny Cove)		2019 / 2020		Ice Lake-U[32] Ice Lake-Y[32]		Ice Lake-SP[33]	
10 nm		Tiger Lake[29] (Willow Cove[34]?)							
10 nm		? (Golden Cove[34])							
7 nm[35]	Unknown								
5 nm[35]									

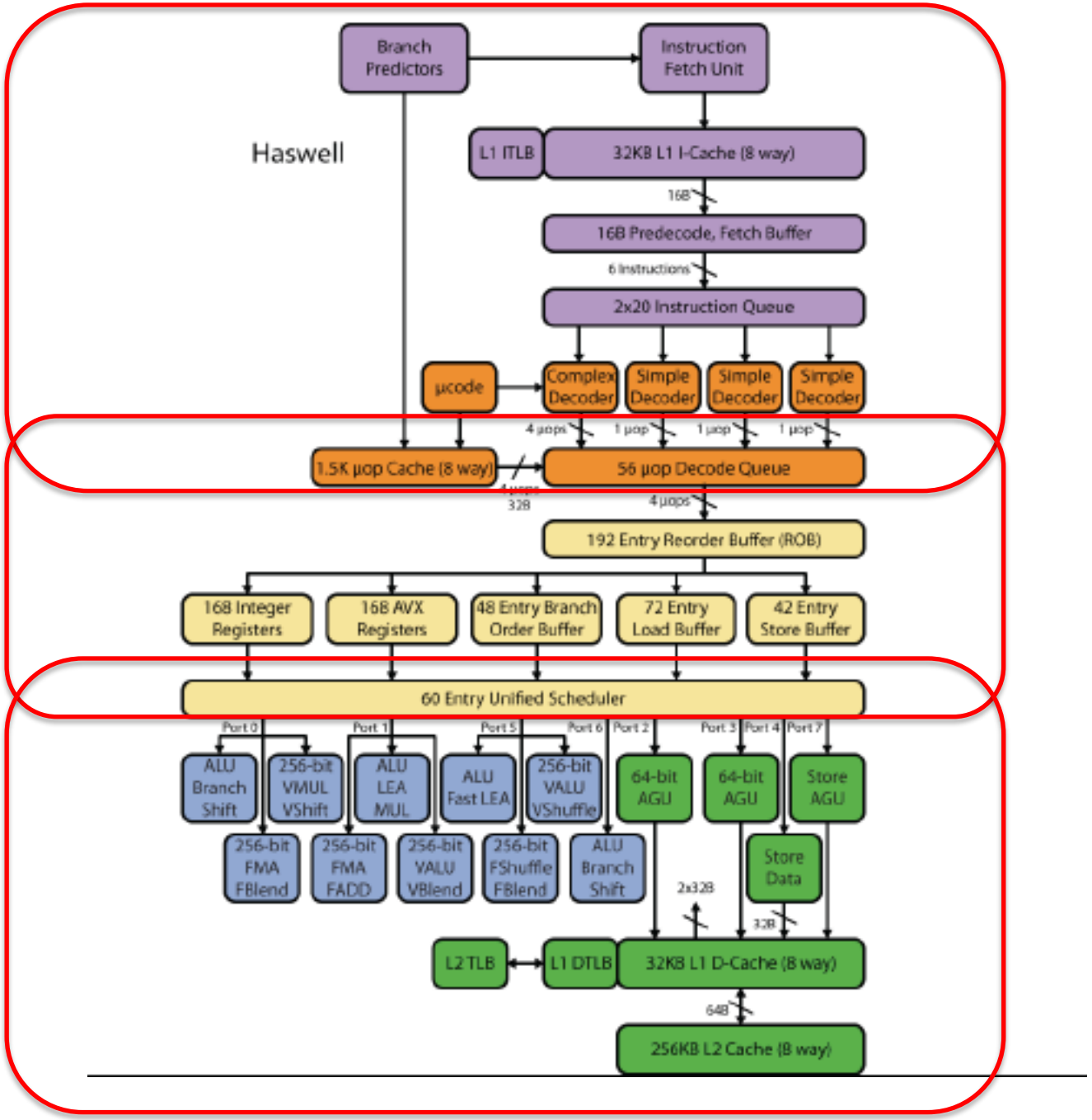
List of Skylake-based Core i9 Processors

Main processor									Memory	
Model	Price	Process	Launched	Cores	Threads	Frequency	Turbo	TDP	Max Mem	Memory Type
i9-7900X	$ 999.00	14 nm	26 June 2017	10	20	3.3 GHz	4.3 GHz	140 W	128 GiB	DDR4-2666
i9-7920X	$ 1,199.00	14 nm	28 August 2017	12	24	2.9 GHz	4.3 GHz	140 W	128 GiB	DDR4-2666
i9-7940X	$ 1,399.00	14 nm	25 September 2017	14	28	3.1 GHz	4.3 GHz	165 W	128 GiB	DDR4-2666
i9-7960X	$ 1,699.00	14 nm	25 September 2017	16	32	2.8 GHz	4.2 GHz	165 W	128 GiB	DDR4-2666
i9-7980XE	$ 1,999.00	14 nm	25 September 2017	18	36	2.6 GHz	4.2 GHz	165 W	128 GiB	DDR4-2666
i9-9990XE		14 nm	3 January 2019	14	28	4 GHz	5 GHz	255 W	128 GiB	DDR4-2666
i9-9820X	$ 898.00	14 nm	November 2018	10	20	3.3 GHz	4.1 GHz	165 W	128 GiB	DDR4-2666
i9-9900X	$ 989.00	14 nm	November 2018	10	20	3.5 GHz	4.4 GHz	165 W	128 GiB	DDR4-2666
i9-9920X	$ 1,189.00	14 nm	November 2018	12	24	3.5 GHz	4.4 GHz	165 W	128 GiB	DDR4-2666
i9-9940X	$ 1,387.00	14 nm	November 2018	14	28	3.3 GHz	4.4 GHz	165 W	128 GiB	DDR4-2666
i9-9960X	$ 1,684.00	14 nm	November 2018	16	32	3.1 GHz	4.4 GHz	165 W	128 GiB	DDR4-2666
i9-9980XE	$ 1,979.00	14 nm	November 2018	18	36	3 GHz	4.4 GHz	165 W	128 GiB	DDR4-2666

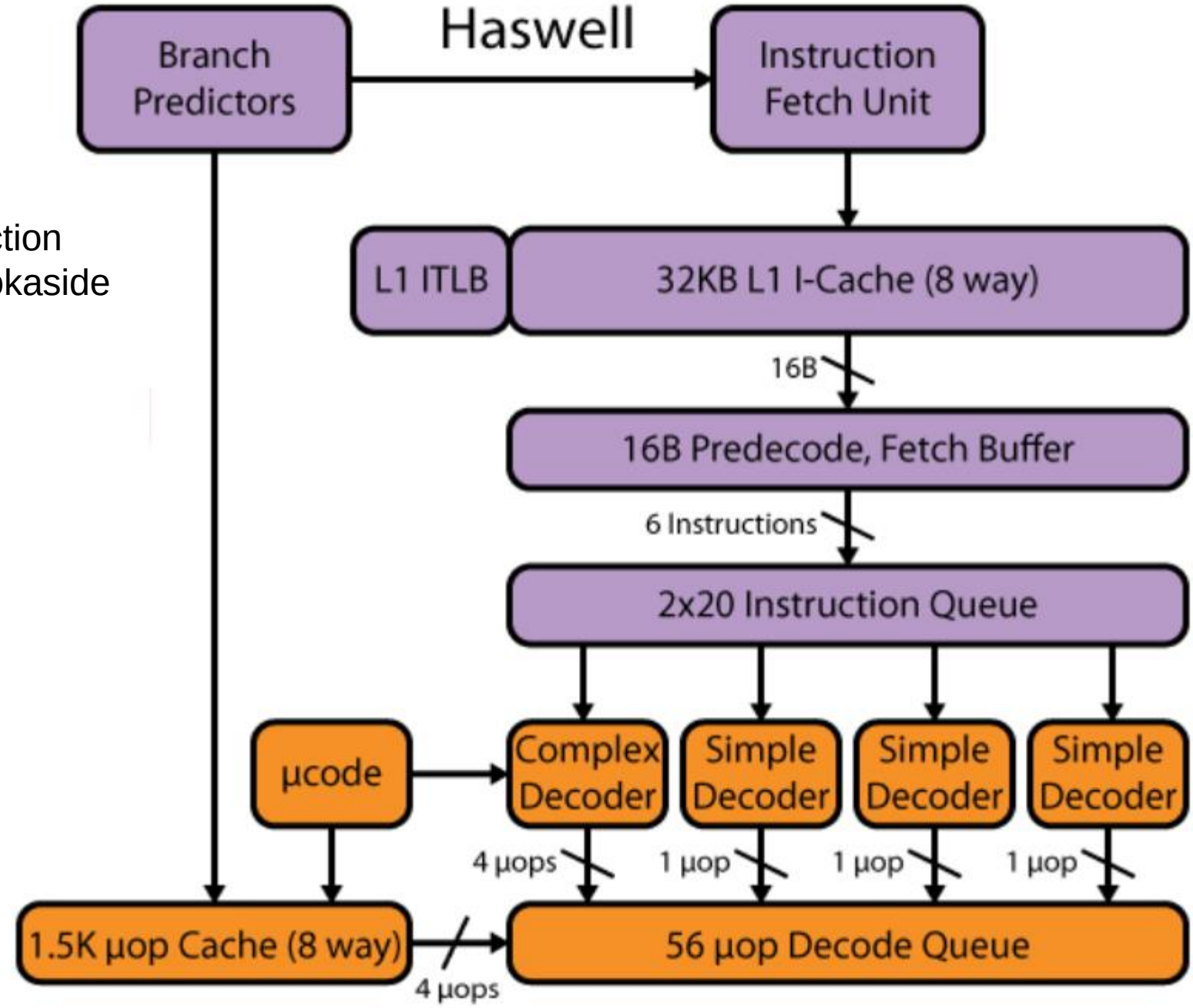
List of Coffee Lake-based Core i9 Processors

Main processor								
Model	Price	Launched	Cores	Threads	Frequency	Turbo	TDP	Max Mem
i9-8950HK	$ 583.00	2 April 2018	6	12	2.9 GHz	4.6 GHz	45 W	64 GiB
i9-9880H			8	16	2.3 GHz	4.8 GHz	45 W	64 GiB
i9-9900			8	16			65 W	64 GiB
i9-9900K	$ 488.00 $ 499.00	19 October 2018	8	16	3.6 GHz		95 W	64 GiB
i9-9900KF	$ 488.00	7 January 2019	8	16	3.6 GHz		95 W	64 GiB
i9-9900T			8	16			35 W	64 GiB
i9-9980HK			8	16	2.4 GHz	5 GHz	45 W	64 GiB

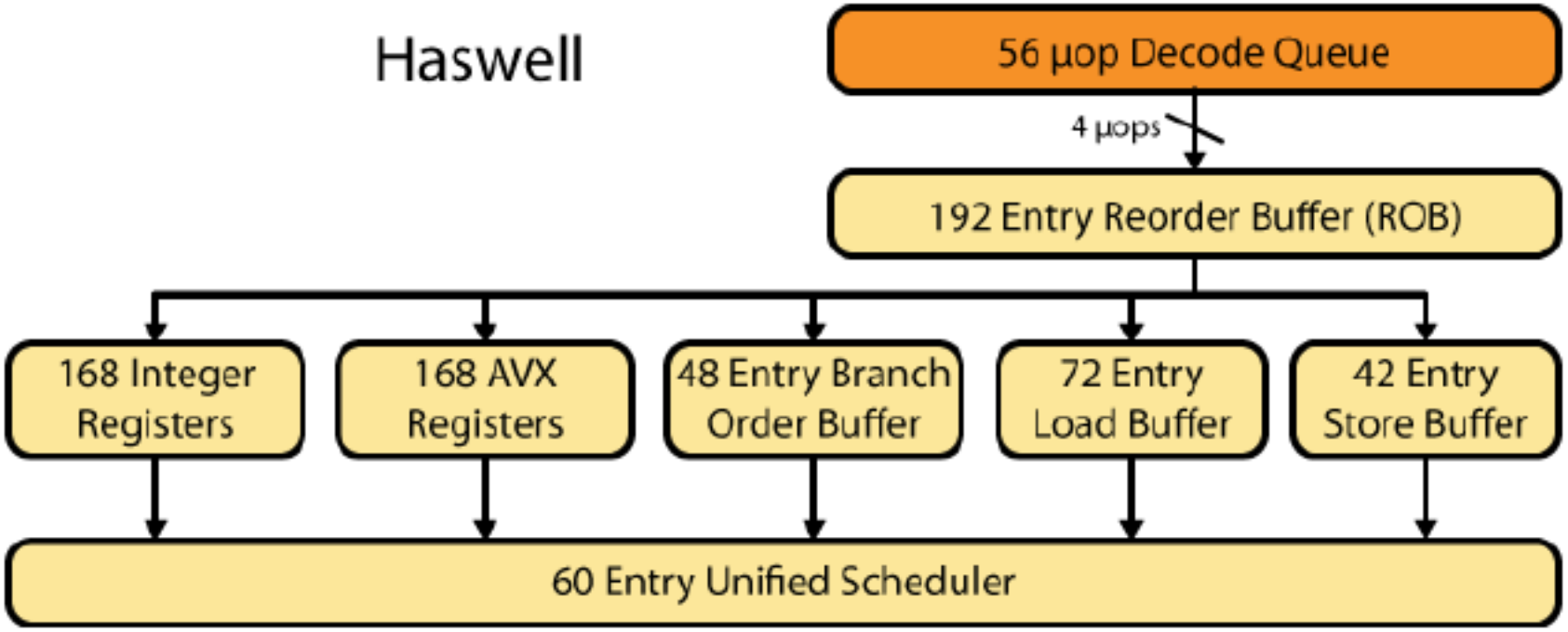
Översikt


Haswell
Branch Predictors
Instruction Fetch Unit
L1 ITLB
32KB L1 I-Cache (8 way)
16B
16B Predecode, Fetch Buffer
6 Instructions
2x20 Instruction Queue
µcode
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
4 µops
1 µop
1 µop
1 µop
1.5K µop Cache (8 way)
56 µop Decode Queue
32B
4 µops
192 Entry Reorder Buffer (ROB)
168 Integer Registers
168 AVX Registers
48 Entry Branch Order Buffer
72 Entry Load Buffer
42 Entry Store Buffer
60 Entry Unified Scheduler
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
ALU Branch Shift
256-bit VMUL VShift
ALU LEA MUL
ALU Fast LEA
256-bit VALU VShuffle
64-bit AGU
64-bit AGU
Store AGU
256-bit FMA FBlend
256-bit FMA FADD
256-bit VALU VBlend
256-bit FShuffle FBlend
ALU Branch Shift
Store Data
2x32B
32B
L2 TLB
L1 DTLB
32KB L1 D-Cache (8 way)
64B
256KB L2 Cache (8 way)

ITLB = instruction translation lookaside buffer



Out-of-order exekvering



Superskalär (Haswell arch.)