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

Äldre arkitektur 32 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 32 bitar

Arkitektur x64

- 16 generella register
  (som jämförelse har 64bits ARM 32 st)
- Ordlängd 64 bitar
- De flesta instruktioner kan jobba mot en operand i register och *en* i minnet (instruktionerna kan givetvis arbeta med bara registerinnehåll också)

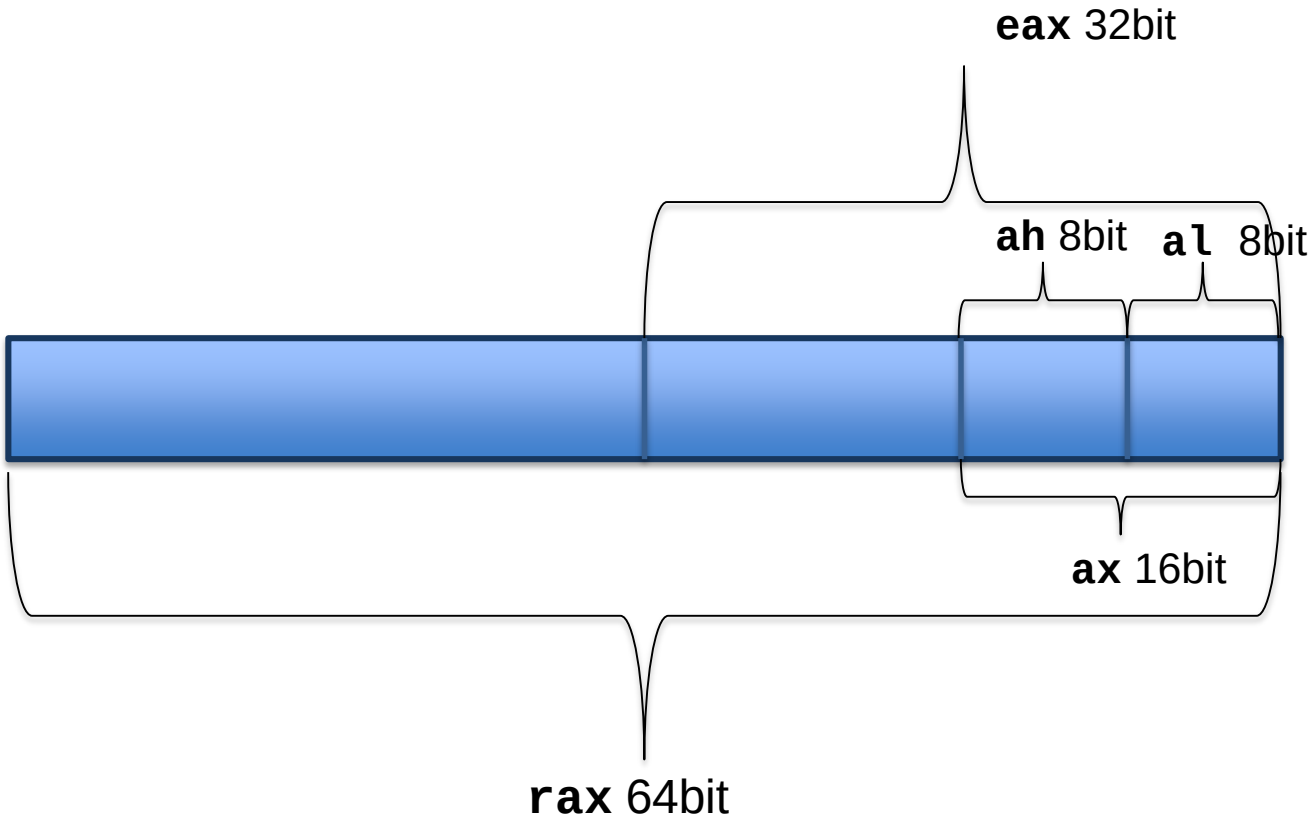
Arkitektur (forts.)

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

64-bits register



Suffix till instruktioner

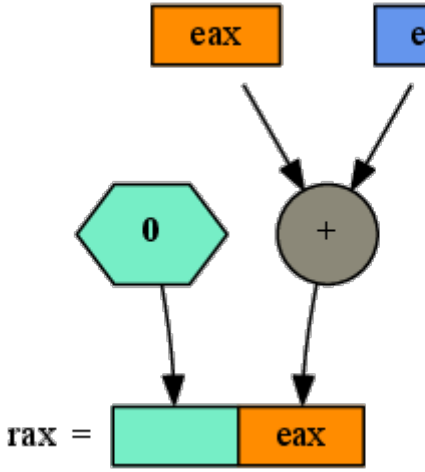
- Suffix till instruktioner anger hur stort dataformat som ska användas
  - b = byte (8 bitar)
  - *s = short (16-bits heltal) eller single (32-bits flyttal)*
  - w = word (16 bitar)
  - l = long (32-bits heltal eller 64-bits flyttal)
  - q = quad (64 bitar)
  - *t = ten bytes (80 bits flyttal)*
- Om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

Några exempel

$ anger immediate data

```
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:



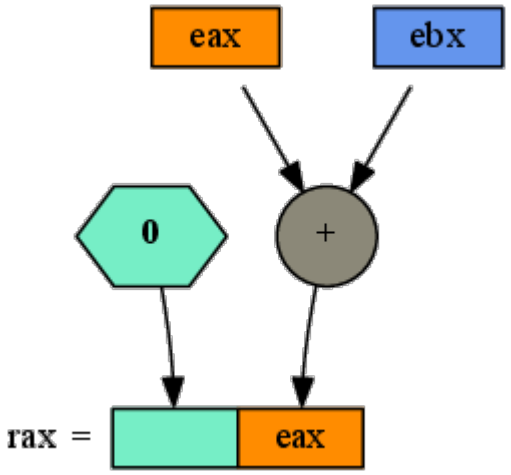

```
addl %ebx,%eax      #eax <- eax+ebx
```

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

formatet måste vara lika stort

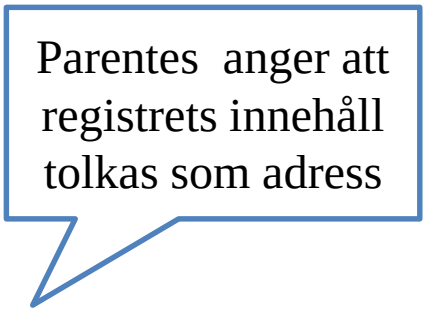
OBS!!!

- 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.

Indirekt adressering till minne

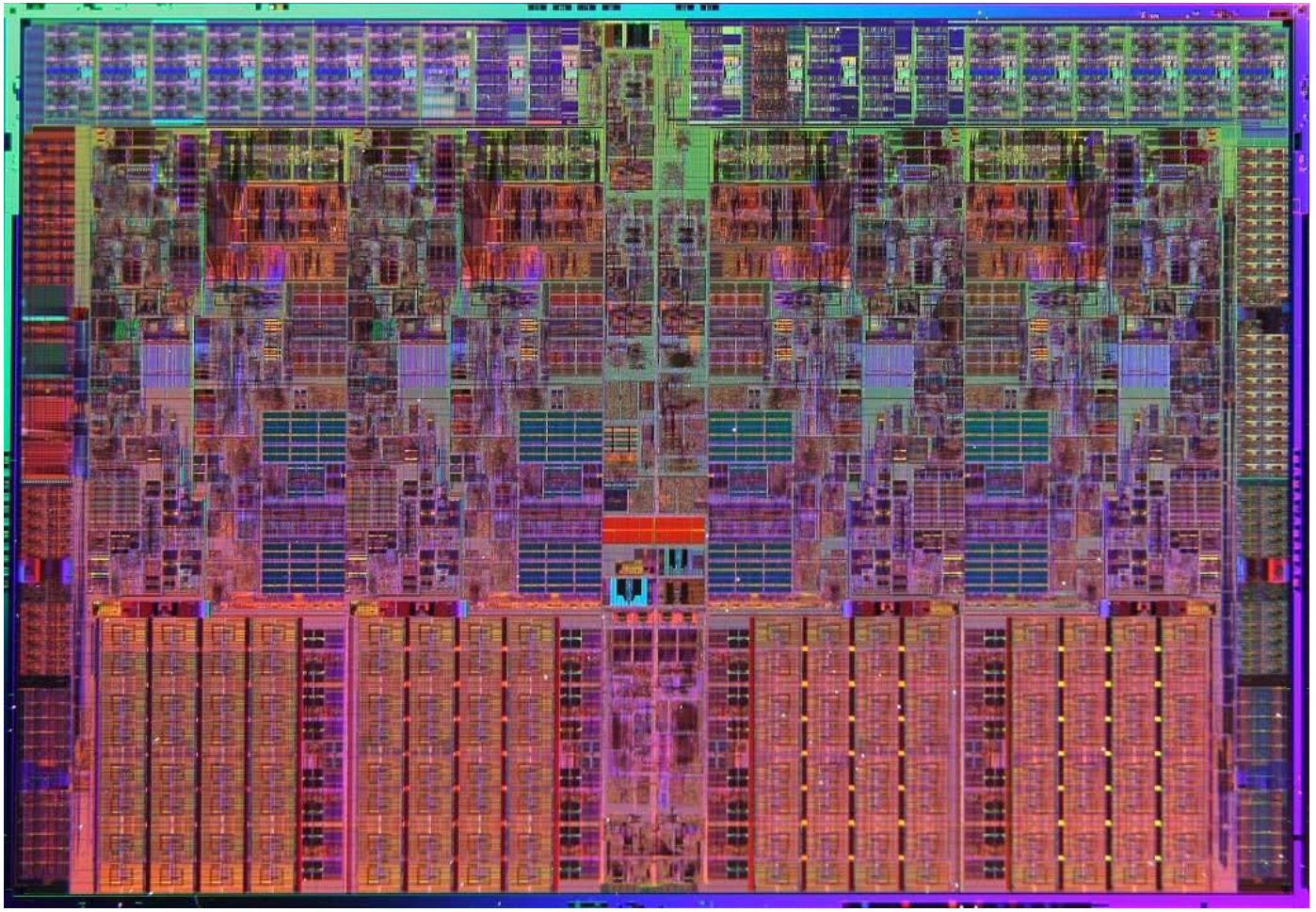



```
movl (%rbx),%eax     #laddar ett 32-bitstal från
                     #minnesadressen rbx pekar på till eax

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

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

Hantering av stacken

- Generella 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

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

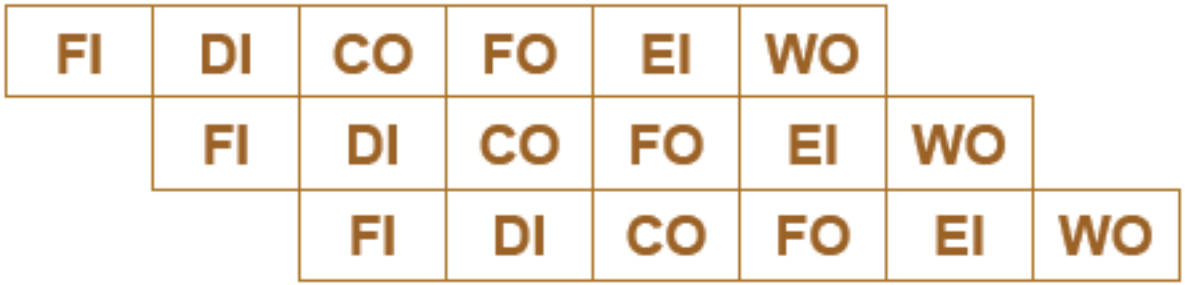
Delayed branching
(repetition från pipelining)

- Om kompilatorn inte hittar en lämplig instruktion att lägga i delay slot, 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*

Principiell pipeline

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



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

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.

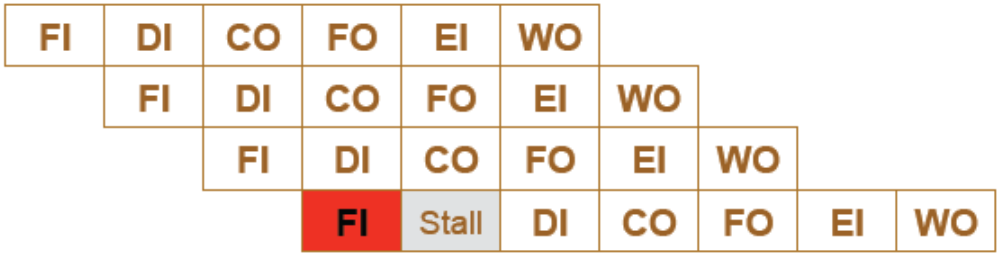
- För villkorliga hopp är det svårare, då måste man veta om hoppet ska tas eller inte

Branch prediction – villkorliga hopp

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

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

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



- Kostnad 1 cykel

- Alternativ 2: Hoppet görs (antagandet var fel)

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

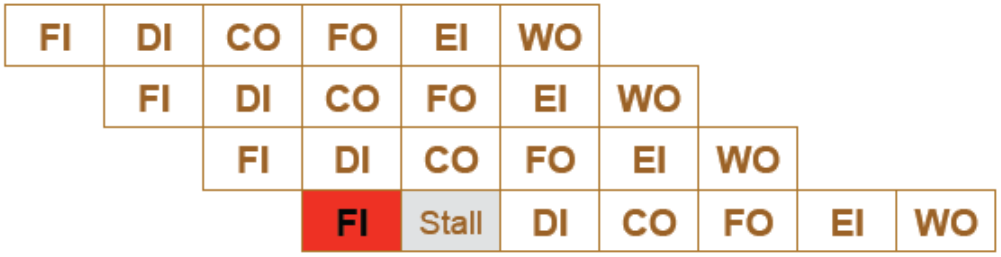
- Kostnad 2 cykler

Branch prediction forts.

- 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

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

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

- Kostnad 2 cykler

Branch prediktion forts.

- Rätt *branch prediction* är viktigt
- 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)

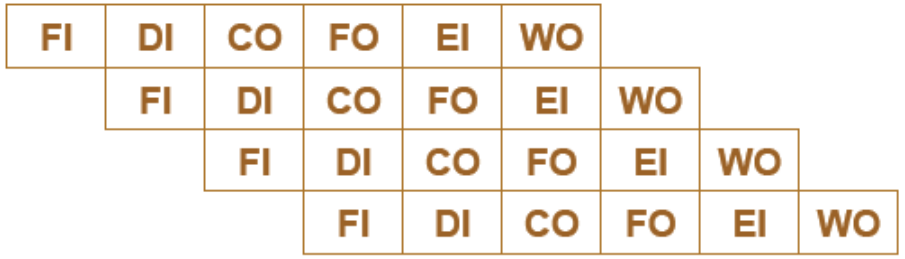
- Strategier för branch prediction:
  - Statisk prediktering
  - Dynamisk prediktering

Spekulativ exekvering forts.

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

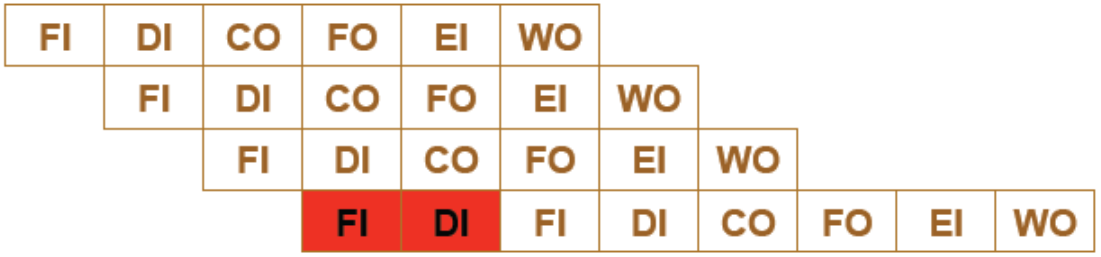
- Alternativ 1: Hoppet görs (antagandet var rätt)

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



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

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



- Kostnad 2 cykler

Statisk *branch prediction*

- Vid statisk branch prediction tas ingen hänsyn till exekveringshistoriken

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

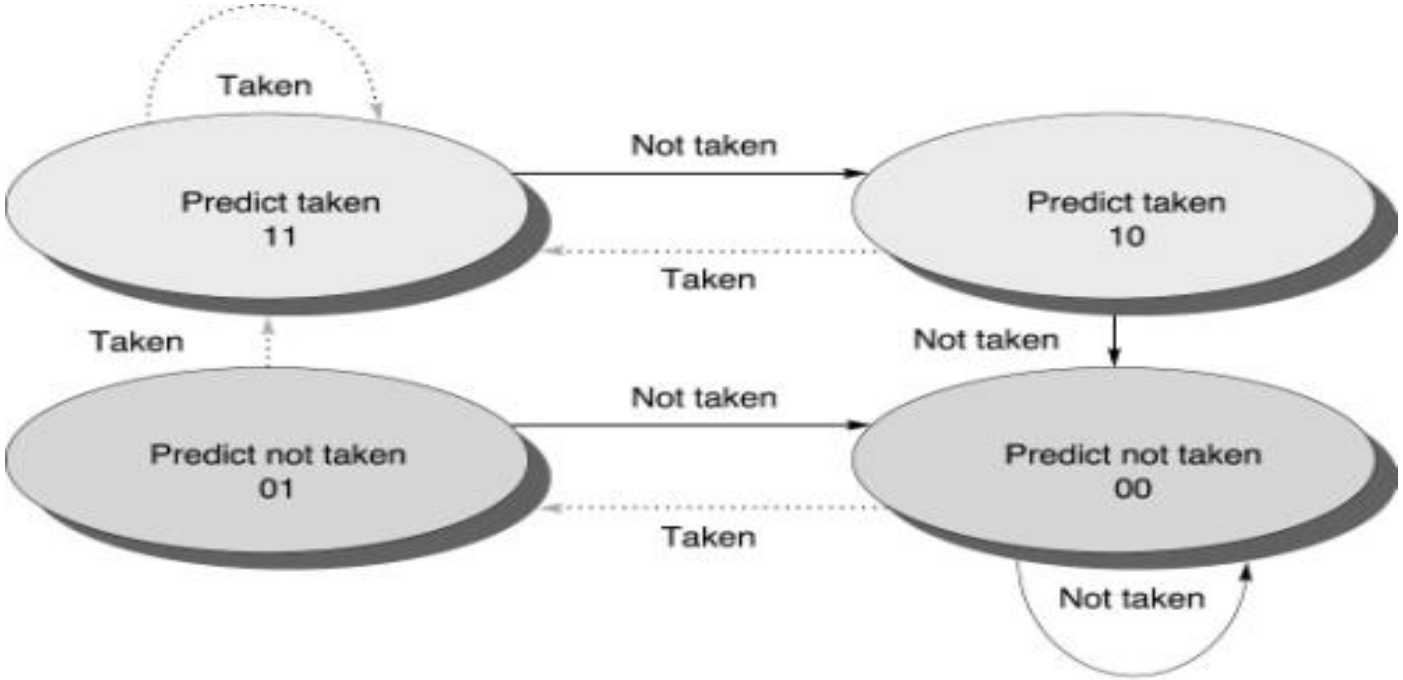
Dynamisk *branch prediction*

- 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.

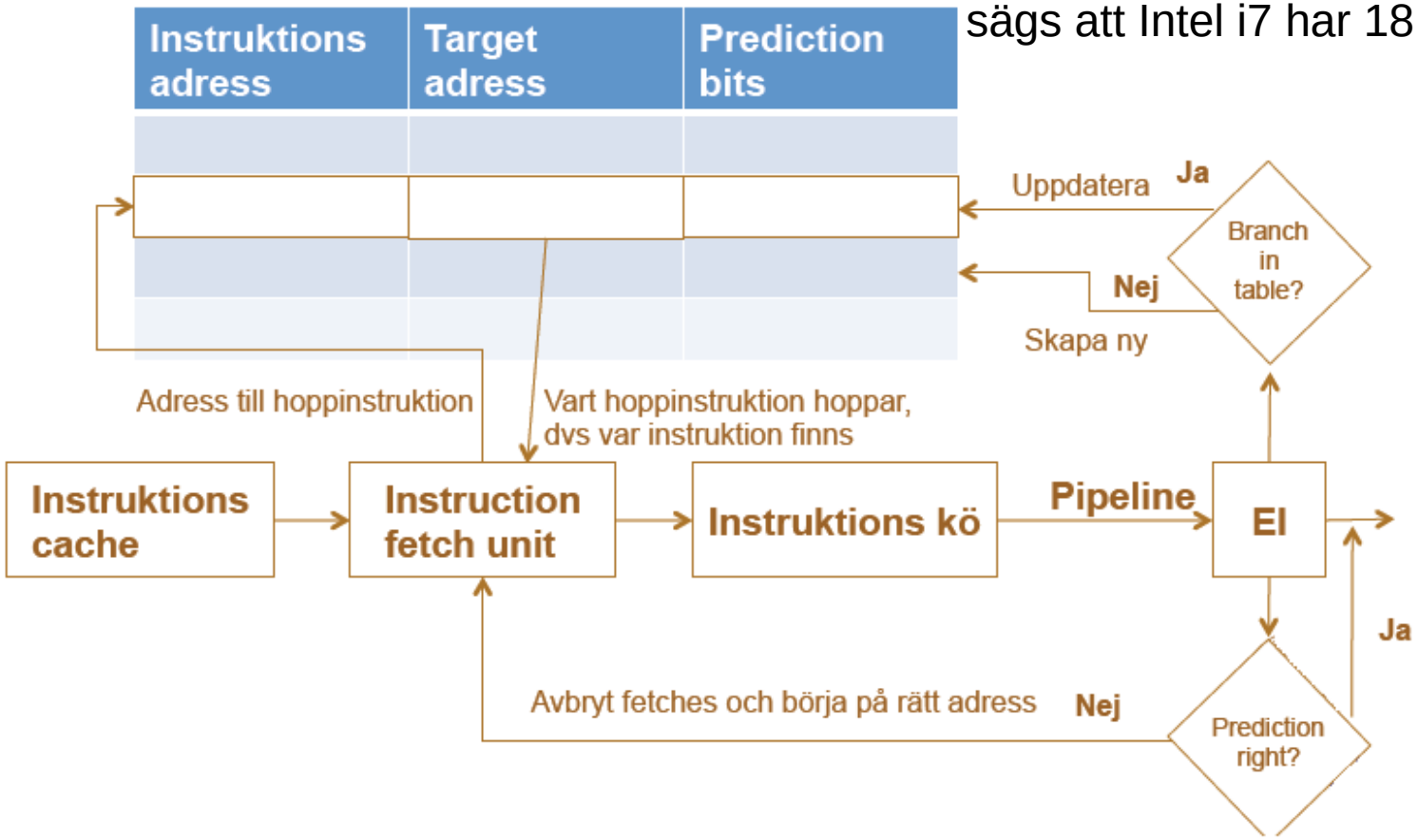
- Man kan använda två bitar för prediktion och på så sätt få en mer ”kvalificerad gissning”

Dynamisk branch prediction, 2 bitar



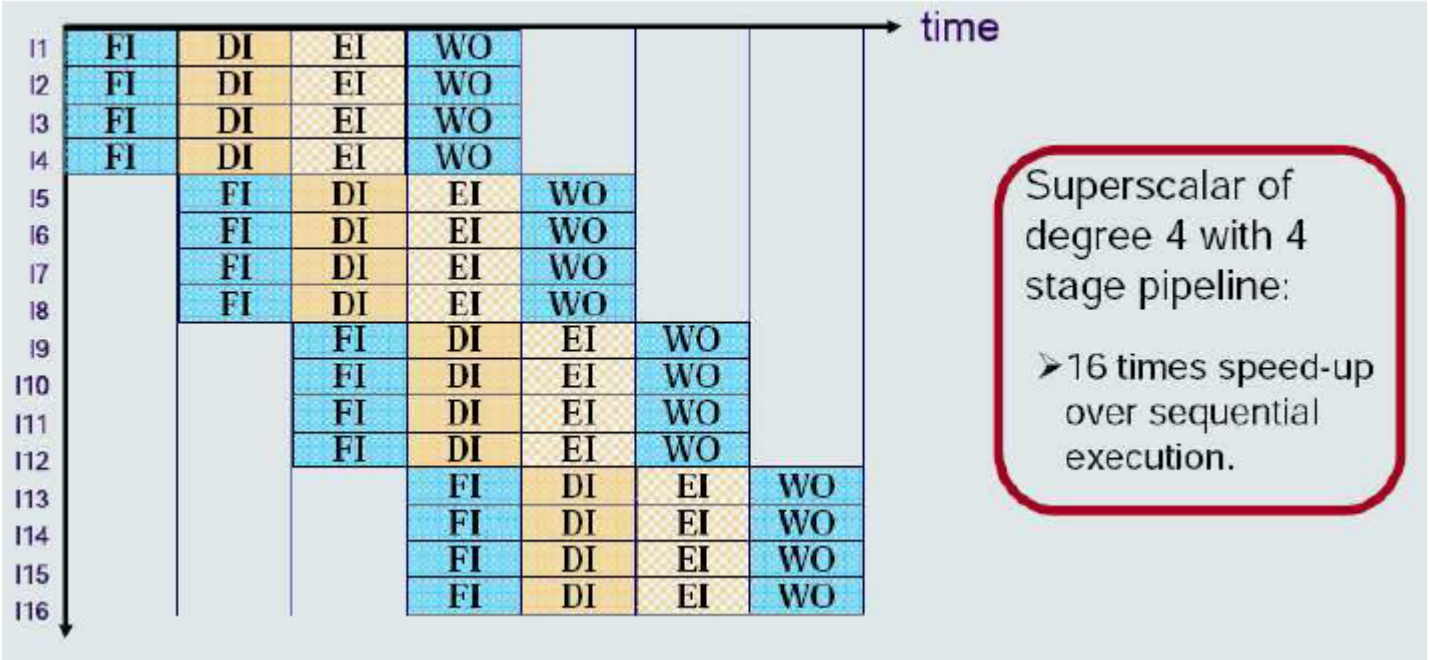
Branch history table (branch target buffer)

Antal bitar flera här, sägs att Intel i7 har 18



Superskalär arkitektur

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



<u>Exempel</u>

- Intel core i7 och AMD Opteron har 4 st

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

Kapacitetsutnyttjande

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

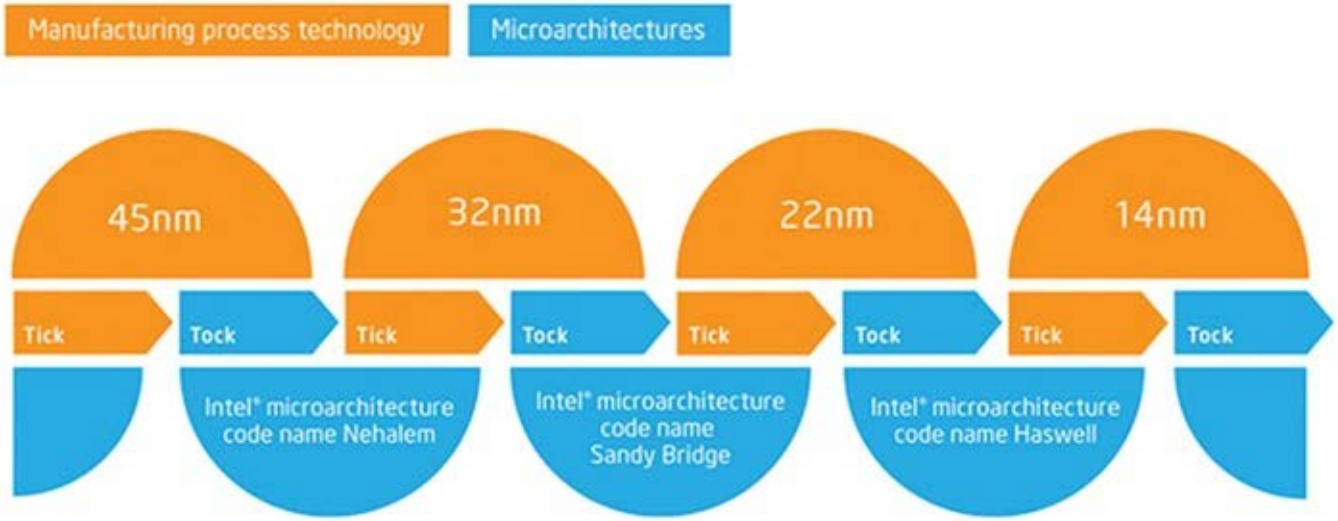
- Ett sätt att utnyttja exekveringskapaciteten bättre är så kallad ”hyperthreading”.
  - Två trådar körs in i strukturen för att kunna fylla pipelinerna bättre (fler oberoende instruktioner att välja på)

- 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 process
- 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	Skylake		2015
Tick	New Process	Skymont		10 nm	2016
Tock	New uArch				2017

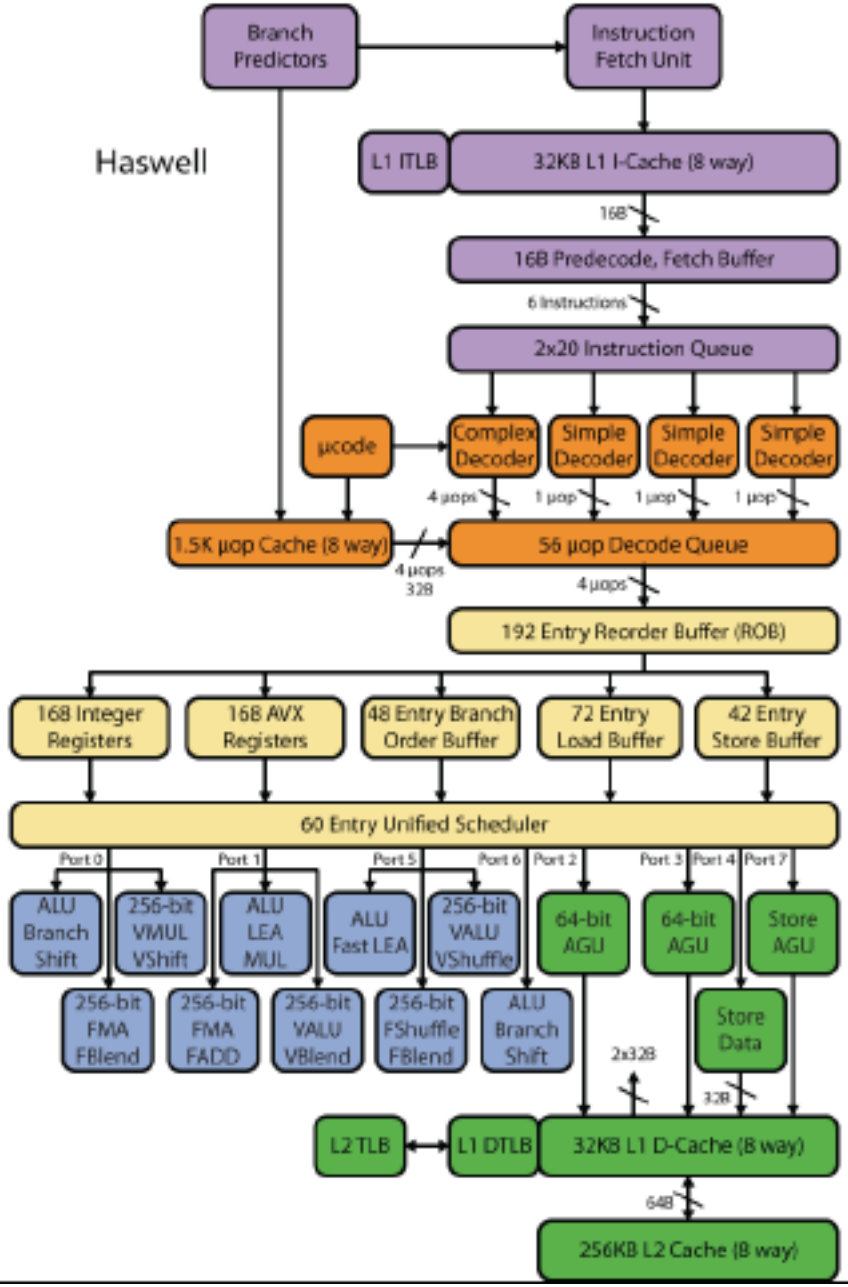


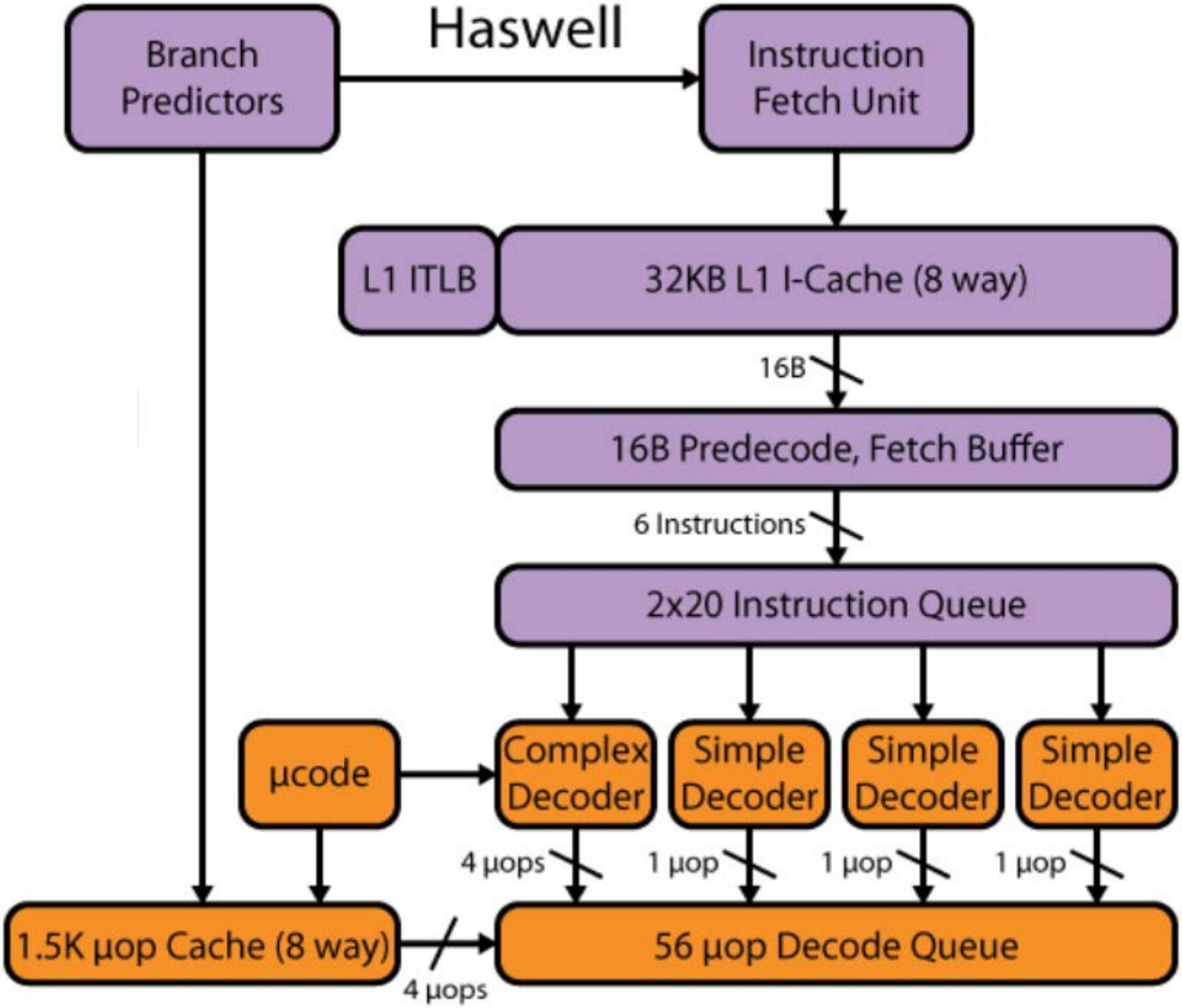
	Tick	Tock	Toe
45nm	Penryn	Nehalem	-
32nm	Westmere	Sandy Bridge	-
22nm	Ivy Bridge	Haswell	Devil's Canyon
14nm	Broadwell	Sky Lake	Kaby Lake
10nm	Cannon Lake (2017)	Ice Lake (2018)	Tiger Lake (2019)

Source: Wikipedia

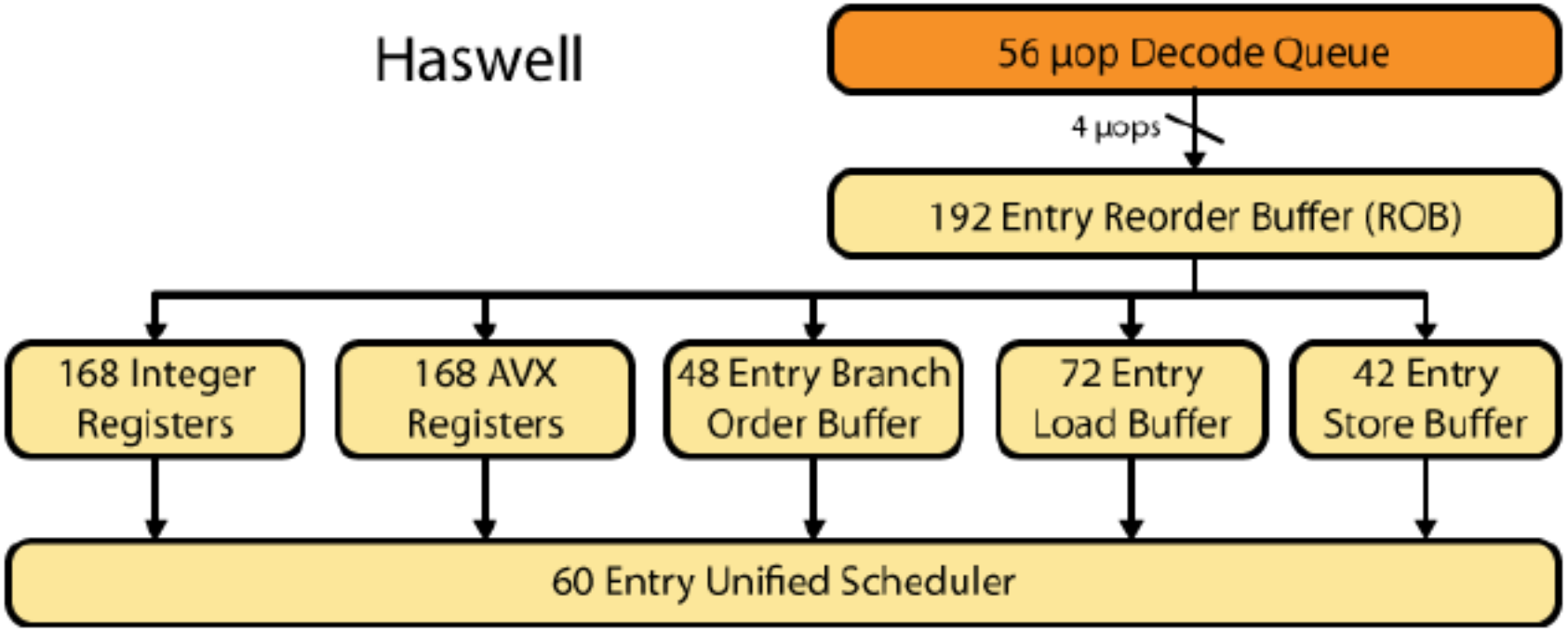
- På sistone verkar det som om Intel arbetar i tre steg istället (toe-steget avser optimering av redan befintlig arkitektur och tillverkningsprocess)

Översikt

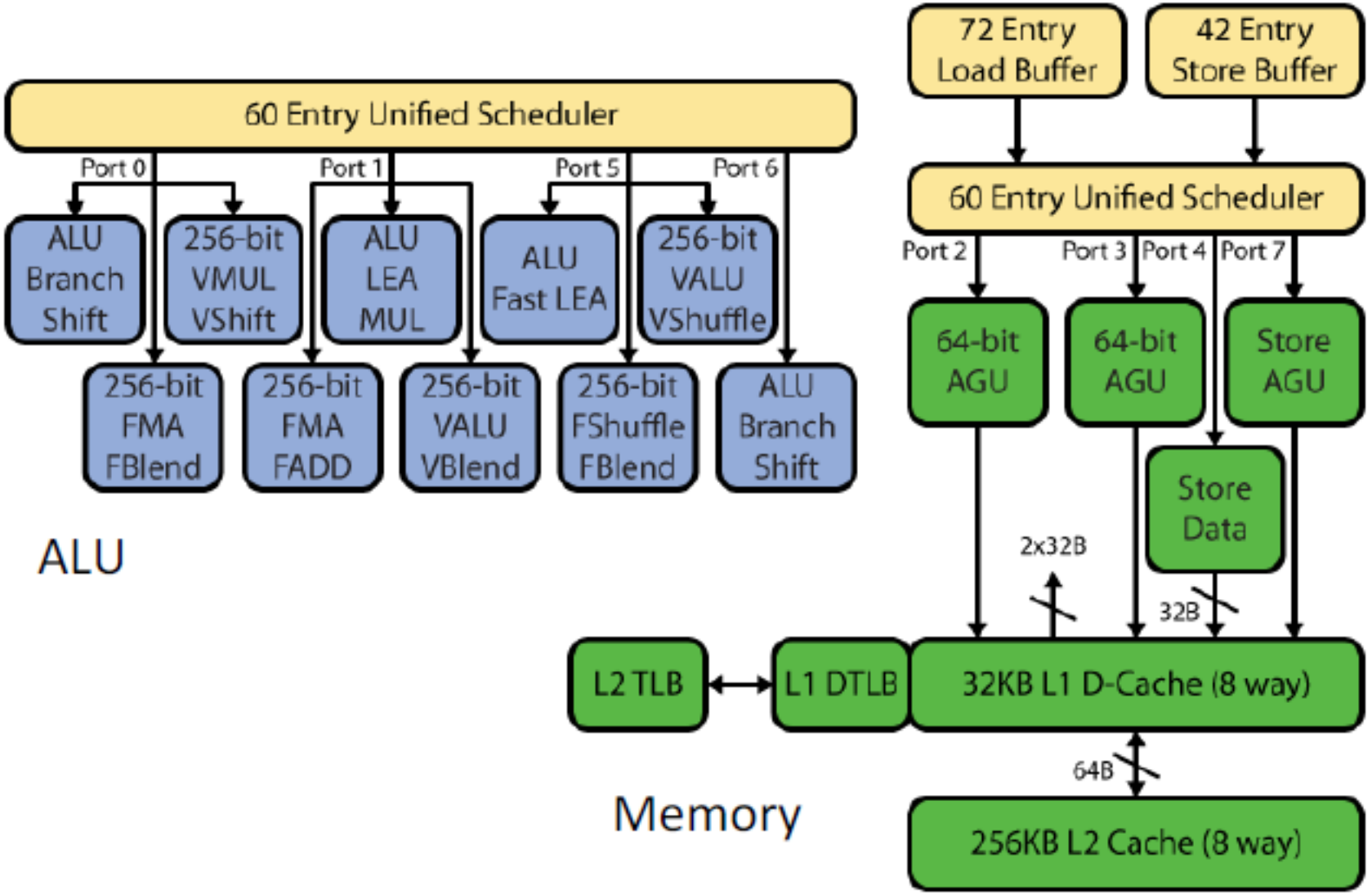

Haswell
Branch Predictors
Instruction Fetch Unit
L1 ITLB
32KB L1 I-Cache (8 way)
16B
16B Predecode, Fetch Buffer
6 Instructions
2x20 Instruction Queue
µcode
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
4 µops
1 µop
1 µop
1 µop
1.5K µop Cache (8 way)
4 µops
32B
56 µop Decode Queue
4 µops
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)


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)
4 µops
56 µop Decode Queue

Out-of-order exekvering



Superskalär (Haswell arch.)



Instruktioner för att flytta data

Instruktion	**Resultat**	**Beskrivning**
`movq` *S, D*	*D* ← *S*	flytta 64-bits ord
`movabsq` *I, R*	*R* ← *I*	flytta 64-bits ord
`movslq` *S, R*	*R* ← **SignExtend**(*S*)	flytta 32-bits ord med teckentillägg
`movsbq` *S, R*	*R* ← **SignExtend**(*S*)	flytta 8-bits ord med teckentillägg
`movzbq` *S, R*	*R* ← **ZeroExtend**(*S*)	flytta 8-bits ord utfyllt med nollor
`pushq` *S*	**R**[%rsp] ← **R**[%rsp] - 8; **M**[**R**[%rsp]] ← *S*	lägg *S* överst på stacken
`popq` *D*	*D* ← **M**[**R**[%rsp]]; **R**[%rsp] ← **R**[%rsp] + 8	hämta till *D* från överst på stacken

Instruktioner för aritmetik och logik

Instruktion	Resultat	Beskrivning
`leaq` *S, D*	*D* ← *&S*	ladda effektiv adress
`incq` *D*	*D* ← *D* + 1	inkrement (räkna upp med 1)
`decq` *D*	*D* ← *D* - 1	dekrement (räkna ned med 1)
`negq` *D*	*D* ← -D	negation (teckenväxling)
`notq` *D*	*D* ← ~D	invertera alla bitar
`addq` *S, D*	*D* ← *D* + *S*	addition
`subq` *S, D*	*D* ← *D* - *S*	subtraktion
`imulq` *S, D*	*D* ← *D* * *S*	multiplikation
`xorq` *S, D*	*D* ← *D* ^ *S*	bitvis xor
`orq` *S, D*	*D* ← *D* | *S*	bitvis eller
`andq` *S, D*	*D* ← *D* & *S*	bitvis och
`salq` *k, D*	*D* ← *D* << *k*	bitvis vänsterskift *k* positioner
`shlq` *k, D*	*D* ← *D* << *k*	samma som ovan
`sarq` *k, D*	*D* ← *D* >> *k*	aritmetiskt högerskift *k* positioner
`shrq` *k, D*	*D* ← *D* >> *k*	logiskt högerskift *k* positioner

Speciella aritmetiska instruktioner

Instruktion	Resultat	Beskrivning
`imulq` *S*	**R**[%rdx]: **R**[%rax] ← *S* × **R**[%rax]	Full multiplikation med teckensatta tal
`mulq` *S*	**R**[%rdx]: **R**[%rax] ← *S* × **R**[%rax]	Full multiplikation med teckenlösa tal
`cltq`	**R**[%rax]← **SignExtend**(**R**[%eax])	Konvertera %eax till 64 bitar
`cqto`	**R**[%rdx]: **R**[%rax] ← **SignExtend**(**R**[%rax])	Konvertera %rax till 128 bitar
`idivq` *S*	**R**[%rdx] ← **R**[%rdx]: **R**[%rax] mod *S* ; **R**[%rax] ← **R**[%rdx]: **R**[%rax] / *S*	Division med teckensatta tal
`divq` *S*	**R**[%rdx] ← **R**[%rdx]: **R**[%rax] mod *S* ; **R**[%rax] ← **R**[%rdx]: **R**[%rax] / *S*	Division med teckenlösa tal

Hoppinstruktioner

Tester att basera villkorliga hopp på

Instruktion	Jämförelse baserad på	Beskrivning
`cmpq` S_2, S_1	$S_1 - S_2$	Jämför 64-bits dataord som teckensatta tal. OBS! Ordningsföljden!
`testq` S_2, S_1	$S_1 \& S_2$	Testar 64-bits dataord

Ovillkorligt hopp

Instruktion	Beskrivning
`jmp` *label*	ovillkorligt hopp till *label*

Villkorliga hopp

Instruktion	Beskrivning
`je` *label*	hoppa om föregående jämförelse är lika med noll
`jne` *label*	hoppa om föregående jämförelse inte är lika med noll
`jg` *label*	hoppa om resultat av jämförelse större än noll
`jl` *label*	hoppa om resultat av jämförelse mindre än noll
`jge` *label*	hoppa om resultat av jämförelse större än eller lika med noll
`jle` *label*	hoppa om resultat av jämförelse mindre än eller lika med noll

Anropskonventioner

- Heltalsparametrar skickas in i **`rdi`**, **`rsi`**, **`rdx`**, **`rcx`**, **`r8`**, **`r9`**
  - första parametern i **`rdi`**
  - andra parametern i **`rsi`**
  - **…**
  - Vid fler än 6 parametrar skickas resten via stacken

- Returvärdet skickas i **`rax`** om heltal

- **`%rbp`**, **`%rbx`** och **`%r12 – %r14`** måste sparas undan och återställas av en subrutin om deras värden förändras i rutinen

Exempelprogram

Datadefinition

```
   .data
str: .asciz ”Fak=%d\n"
buf: .asciz "xxxxxxxxx”
endTxt: .asciz ”slut\n”
```

Exempelprogram

```
    .text
    .global main
main:
    pushq $0        #Stacken ska vara 16 bytes ”aligned”
    movq $5, %rdi   # Beräkna 5!
    call fac
    movq %rax, %rsi #Flytta returvärdet till argumentregistret
    movq $str, %rdi # skriv ut Fak= ”resultat”
    call printf
# läs med fgets(buf,5,stdin)
    movq $buf, %rdi    # lägg i buf
    movq $5,%rsi       # högst 5-1=4 tecken
    movq stdin, %rdx   # från standard input
    call fgets
    movq $buf, %rdi
    call printf        # skriv ut buffert
    movq $endTxt, %rdi # följd av slut
    call printf
    call exit          # avsluta programmet
```

Exempelprogram forts

```
# Här finns funktionen n! (rekursiv)
fac:
    cmpq $1,%rdi   # if n>1
    jle LABEL
    pushq %rdi     #lägg anropsvärde på stacken
    decq %rdi      #räkna ned värdet med 1
    call fac       #temp = fakultet av (n-1)
    popq %rdi      #hämta från stack
    imul %rdi,%rax # return n*temp
    ret # Återvänd
LABEL:
    movq $1,%rax   # else return 1
    ret # Återvänd
```