- RISC-V ukratko
  - Organizovan u 3 osnovna seta naredbi za podršku za 32, odnosno 64 bita
    - RV32I - osnovni 32-bitni set sa 32 registra
    - RV32E - osnovni 32-bitni set sa 16 registara; namenjen za embedded primene
    - RV64I - osnovni 64-bitni set naredbi, sa 64-bitnim registrima
    - Dosta proširenja osnovnih setova:
      - M - celobrojno množenje i deljenje
      - A - atomske naredbe za konkurentno programiranje
      - F - 32-bitni pokretni zarez, sa 32 FP registra
      - D - 64-bitno proširenje pokretnog zareza
      - Q - 128-bitno proširenje pokretnog zareza
      - L - 128 i 64-bitni decimalni FP
      - C - kompresovana verzija seta naredbi, za mali utrošak memorije
    - Koristićemo RV64IMAFD

- RISC-V ukratko
  - 32 64-bitna registra opšte namene x0,...,x31
    - x0 uvek ima vrednost 0!
  - 32 FP registra f0,...,f31 koji mogu operisati sa 32 ili 64-bitnim vrednostima
  - tipovi podataka
    - celobrojni: 8,16,32,64 bita
    - FP: 32,64 bita
  - Celobrojne operacije uvek rade sa 64 bita
    - 8,16,32-bitne celobrojne vrednosti se prilikom smeštanja u registre proširuju na 64 bita
      - kako se proširuju neoznačeni, a kako označeni brojevi?

- RISC-V ukratko
  - Načini adresiranja
    - neposredno (immediate), za konstante
    - sa udaljenjem (displacement), pri čemu je za udaljenje rezervisano 12 bita
      - slično labela(%registar) kod x86
      - ako se za udaljenje stavi 0, dobija se indirektno adresiranje
      - ako se koristi x0, dobija se (ograničeno) apsolutno adresiranje
    - svaki bajt ima svoju adresu, adrese su 64-bitne
    - višebajtne vrednosti se smeštaju u little-endian formatu (kao x86), najmanje značajan deo na najnižoj adresi
  - Vrste naredbi
    - aritmetičke, logičke - nad registrima (R tip)
    - punjenje iz memorije, konstante (I tip)
    - smeštanje u memoriju, poređenje, grananje (S tip)
    - pozivi potprograma (U tip)

- RISC-V ukratko
  - Vrste naredbi - objašnjenje skraćenica:
    - subskript označava selekciju bita, npr. $Regs[x4]_0$ je najznačajniji bit x4
      - kod strelice znači broj bita koji se prenose
    - superskript označava replikaciju bita, npr. 0^{48} znači 48 binarnih nula
    - ## se koristi za spajanje bita, a ≪ i ≫ su operacije pomeranja (shift) bita u levo, odnosno desno

- RISC-V ukratko
  - Vrste naredbi (najznačajniji bit je 0!)

Example instruction	Instruction name	Meaning
`ld x1,80(x2)`	Load doubleword	Regs[x1] ← Mem[80+Regs[x2]]
`lw x1,60(x2)`	Load word	Regs[x1] $\leftarrow_{64}$ Mem[60+Regs[x2]]$_0$)^{32} ## Mem[60+Regs[x2]]
`lwu x1,60(x2)`	Load word unsigned	Regs[x1] $\leftarrow_{64}$ 0^{32} ## Mem[60+Regs[x2]]
`lb x1,40(x3)`	Load byte	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{56} ## Mem[40+Regs[x3]]
`lbu x1,40(x3)`	Load byte unsigned	Regs[x1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[x3]]
`lh x1,40(x3)`	Load half word	Regs[x1] $\leftarrow_{64}$ (Mem[40+Regs[x3]]$_0$)^{48} ## Mem[40+Regs[x3]]
`flw f0,50(x3)`	Load FP single	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x3]] ## 0^{32}
`fld f0,50(x2)`	Load FP double	Regs[f0] $\leftarrow_{64}$ Mem[50+Regs[x2]]
`sd x2,400(x3)`	Store double	Mem[400+Regs[x3]] $\leftarrow_{64}$ Regs[x2]
`sw x3,500(x4)`	Store word	Mem[500+Regs[x4]] $\leftarrow_{32}$ Regs[x3]$_{32..63}$
`fsw f0,40(x3)`	Store FP single	Mem[40+Regs[x3]] $\leftarrow_{32}$ Regs[f0]$_{0..31}$
`fsd f0,40(x3)`	Store FP double	Mem[40+Regs[x3]] $\leftarrow_{64}$ Regs[f0]
`sh x3,502(x2)`	Store half	Mem[502+Regs[x2]] $\leftarrow_{16}$ Regs[x3]$_{48..63}$
`sb x2,41(x3)`	Store byte	Mem[41+Regs[x3]] $\leftarrow_{8}$ Regs[x2]$_{56..63}$

- RISC-V ukratko
  - Vrste naredbi

Example instrucmtion	Instruction name	Meaning
`add x1,x2,x3`	Add	`Regs[x1]←Regs[x2]+Regs[x3]`
`addi x1,x2,3`	Add immediate unsigned	`Regs[x1]←Regs[x2]+3`
`lui x1,42`	Load upper immediate	$Regs[x1] \leftarrow 0^{32} \# \# 42 \# \# 0^{12}$
`sll x1,x2,5`	Shift left logical	`Regs[x1]←Regs[x2]<<5`
`slt x1,x2,x3`	Set less than	`if (Regs[x2]<Regs[x3]) Regs[x1]←1 else Regs[x1]←0`

  - lui - konstruisanje 32-bitnih konstanti (viših 20 bita)

- RISC-V ukratko
  - Vrste naredbi

Example instruction	Instruction name	Meaning
`jal x1,offset`	Jump and link	`Regs[x1]←PC+4; PC←PC+(offset<<1)`
`jalr x1,x2,offset`	Jump and link register	`Regs[x1]←PC+4; PC←Regs[x2]+offset`
`beq x3,x4,offset`	Branch equal zero	`if(Regs[x3]==Regs[x4]) PC←PC+(offset<<1)`
`bgt x3,x4,name`	Branch not equal zero	`if(Regs[x3]>Regs[x4]) PC←PC+(offset<<1)`

  - jal - skače na relativni 20-bitni offset*2, povratnu adresu ostavlja u zadati registar
    - krajnja adresa mora biti deljiva sa 4
    - ako povratna adresa ne treba, za prvi reg. se stavi x0, što predstavlja šta?
  - jalr - širi adresni opseg (32 bita)
    - ako se za povratnu adresu stavi x0, za registar za skok se iskoristi ranije postavljena povratna adresa, a ofset se postavi na nulu, dobije se?

- RISC-V ukratko

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP; only memory address mode is 12-bit displacement+contents of a GPR*
`lb, lbu, sb`	Load byte, load byte unsigned, store byte (to/from integer registers)
`lh, lhu, sh`	Load half word, load half word unsigned, store half word (to/from integer registers)
`lw, lwu, sw`	Load word, store word (to/from integer registers)
`ld, sd`	Load doubleword, store doubleword
Arithmetic/logical	*Operations on data in GPRs. Word versions ignore upper 32 bits*
`add, addi, addw, addiw, sub, subi, subw, subiw`	Add and subtract, with both word and immediate versions
`slt, sltu, slti, sltiu`	set-less-than with signed and unsigned, and immediate
`and, or, xor, andi, ori, xori`	and, or, xor, both register-register and register-immediate
`lui`	Load upper immediate: loads bits 31..12 of a register with the immediate value. Upper 32 bits are set to 0
`auipc`	Sums an immediate and the upper 20-bits of the PC into a register; used for building a branch to any 32-bit address
`sll, srl, sra, slli, srli, srai, sllw,slliw, srli, srliw, srai, sraiw`	Shifts: logical shift left and right and arithmetic shift right, both immediate and word versions (word versions leave the upper 32 bit untouched)
`mul, mulw, mulh, mulhsu, mulhu, div,divw, divu, rem, remu, remw, remuw`	Integer multiply, divide, and remainder, signed and unsigned with support for 64-bit products in two instructions. Also word versions

- RISC-V ukratko
  - Vrste naredbi

Control	*Conditional branches and jumps; PC-relative or through register*
`beq, bne, blt, bge, bltu, bgeu`	Branch based on compare of two registers, equal, not equal, less than, greater or equal, signed and unsigned
`jal,jalr`	Jump and link address relative to a register or the PC
Floating point	*All FP operation appear in double precision (.d) and single (.s)*
`flw, fld, fsw, fsd`	Load, store, word (single precision), doubleword (double precision)
`fadd, fsub, fmult, fiv, fsqrt, fmadd, fmsub, fnmadd, fnmsub, fmin, fmax, fsgn, fsgnj, fsjnx`	Add, subtract, multiply, divide, square root, multiply-add, multiply-subtract, negate multiply-add, negate multiply-subtract, maximum, minimum, and instructions to replace the sign bit. For single precision, the opcode is followed by: .s, for double precision: .d. Thus fadd.s, fadd.d
`feq, flt, fle`	Compare two floating point registers; result is 0 or 1 stored into a GPR
`fmv.x.*, fmv.*.x`	Move between the FP register abd GPR, "*" is s or d
`fcvt.*.l, fcvt.l.*, fcvt.*.lu, fcvt.lu.*, fcvt.*.w, fcvt.w.*, fcvt.*.wu, fcvt.wu.*`	Converts between a FP register and integer register, where "*" is S or D for single or double precision. Signed and unsigned versions and word, doubleword versions

- Paralelizam unutar procesora
  - Korišćenje ILP u kompajlerima
    - za puno korišćenje ILP, treba naći sekvence nepovezanih naredbi kojima se može napuniti protočna struktura i koje se mogu paralelno izvršiti
    - zavisne naredbe moraju biti dovoljno razdvojene da ne dođe do zastoja
    - uspešnost raspoređivanja zavisi od broja naredbi koje mogu biti preklopljene i kašnjenja koje uvodi izvršavanje zavisnih naredbi; pretpostavljaju se ova kašnjenja:

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

- Paralelizam unutar procesora
  - Korišćenje ILP u kompajlerima
    - pretpostavlja se protočna struktura sa 5 stepeni i da grananja imaju kašnjenje od 1 ciklusa
    - primer:

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

      - pojedinačne iteracije su međusobno nezavisne
    - RISC-V verzija:
      - x1 - adresa poslednjeg elementa, f2 - vrednost s
      - x2 - adresa prvog elementa umanjena za 8 (zašto?)

```
Loop: fld    f0,0(x1)     f0 <- element
      fadd.d f4,f0,f2     f4 <- f0+f2
      fsd    f4,0(x1)     element <- f4
      addi   x1,x1,-8     prethodni element niza
      bne    x1,x2,Loop   skok ako je x1!=x2
```

- Paralelizam unutar procesora
  - Korišćenje ILP u kompajlerima
    - po ciklusima (broj na kraju je redni broj ciklusa):

```
Loop: fld    f0,0(x1)     1
      zastoj              2
      fadd.d f4,f0,f2     3
      zastoj              4
      zastoj              5
      fsd    f4,0(x1)     6
      addi   x1,x1,-8     7
      bne    x1,x2,Loop   8
```

    - premeštanje addi naredbe (da li može?), 1 ciklus manje:

```
Loop: fld    f0,0(x1)
      addi   x1,x1,-8
      fadd.d f4,f0,f2
      zastoj
      zastoj
      fsd    f4,8(x1)
      bne    x1,x2,Loop
```

      - svaka iteracija je sada 7 ciklusa umesto 8

- Paralelizam unutar procesora
  - Korišćenje ILP u kompajlerima
    - loop-unrolling sa 4 tela petlje (26 ciklusa, 6500 ukupno)

```
Loop: fld    f0,0(x1)     f0 <- element
      fadd.d f4,f0,f2     f4 <- f0+f2
      fsd    f4,0(x1)     element <- f4
      fld    f6,-8(x1)    f6 <- element-1
      fadd.d f8,f6,f2     f8 <- f6+f2
      fsd    f8,-8(x1)    element-1 <- f8
      fld    f10,-16(x1)  f10 <- element-2
      fadd.d f12,f10,f2   f12 <- f10+f2
      fsd    f12,-16(x1)  element-2 <- f12
      fld    f14,-24(x1)  f14 <- element-3
      fadd.d f16,f14,f2   f16 <- f14+f2
      fsd    f16,-24(x1)  element-3 <- f16
      addi   x1,x1,-32    umanji za 4 emementa
      bne    x1,x2,Loop   skok ako x1 != x2
```

      - kakva adresa sada mora biti u x2?

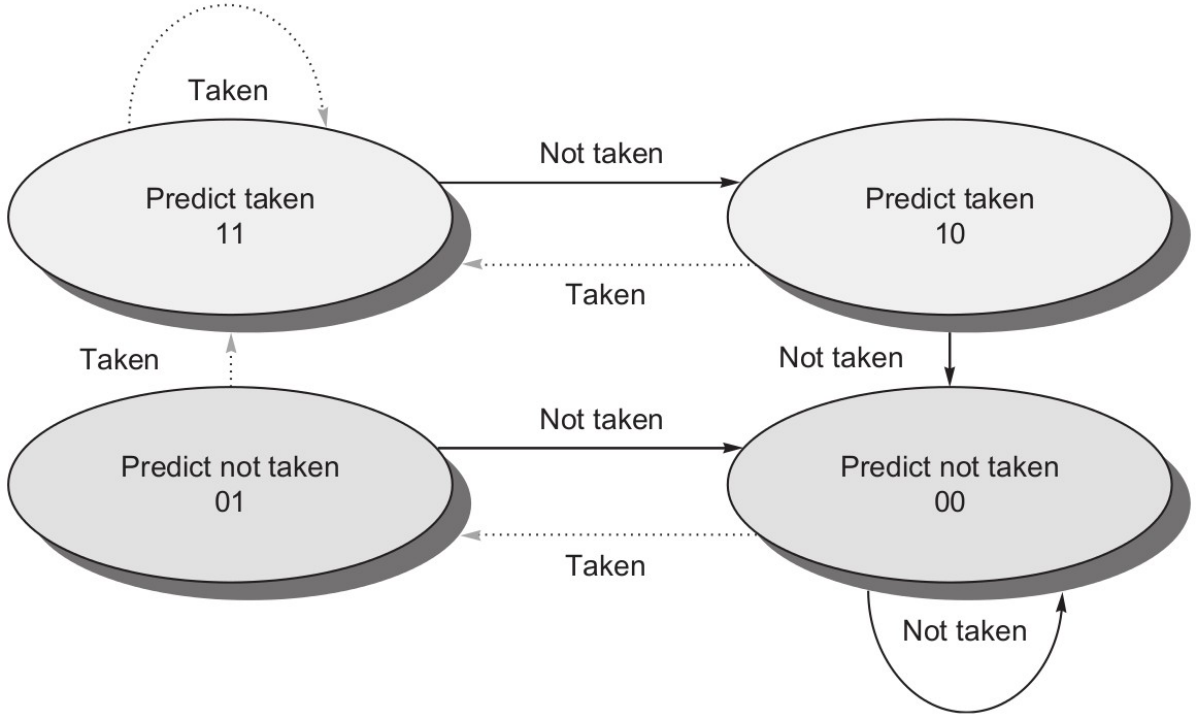
- Paralelizam unutar procesora
  - Korišćenje ILP u kompajlerima
    - optimizacija preplitanja (14 ciklusa, 3500 ukupno)

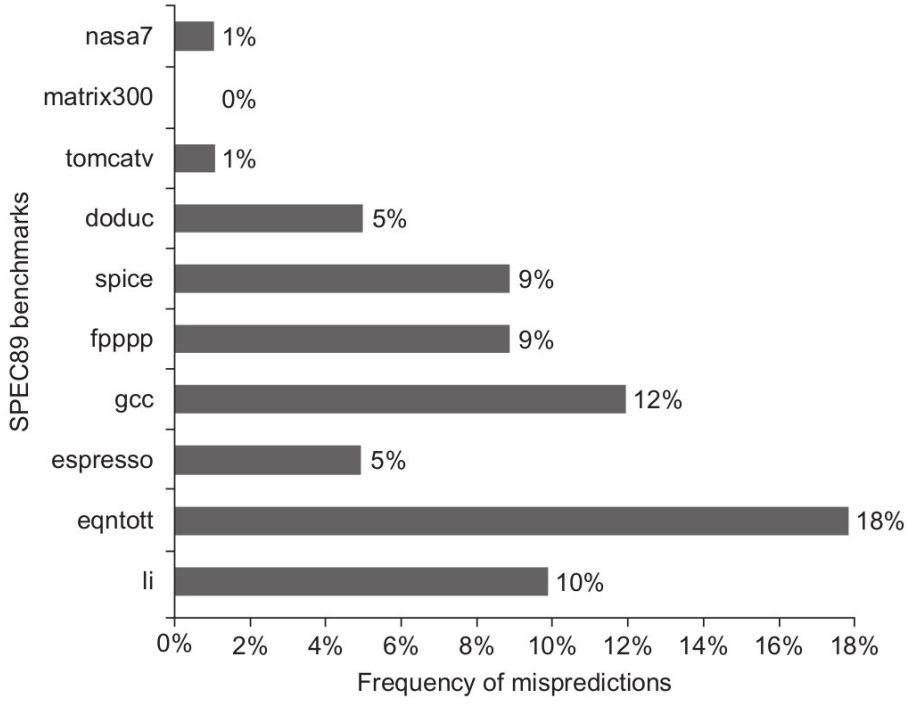
```
Loop: fld    f0,0(x1)
      fld    f6,-8(x1)
      fld    f10,-16(x1)
      fld    f14,-24(x1)
      fadd.d f4,f0,f2
      fadd.d f8,f6,f2
      fadd.d f12,f10,f2
      fadd.d f16,f14,f2
      fsd    f4,0(x1)
      fsd    f8,-8(x1)
      fsd    f12,-16(x1)
      fsd    f16,-24(x1)
      addi   x1,x1,-32
      bne    x1,x2,Loop
```

      - loop-unrolling + preraspodela -> sa 8000 na 3500

- Paralelizam unutar procesora
  - Korišćenje ILP u kompajlerima
    - primer petlje - šta je sve urađeno?
      - utvrđeno je da su međusobne iteracije petlje nezavisne
      - iskorišteni su različiti registri za odmotane iteracije
      - izbačena su nepotrebna poređenja i skokovi
      - utvrđeno je da su load/store operacije pojedinih iteracija nakon toga nezavisne
      - izmenjen je redosled izvršavanja naredbi da se izbegnu zastoji u protočnoj strukturi i obezbedi mnogo bolje paralelno izvršavanje
    - loop-unrolling dobici i gubici
      - smanjen broj naredbi za samu petlju u okviru jedne iteracije
        - u finalnoj varijanti, 2 od 14 ciklusa se troše na samu petlju (addi i bne), što dođe 1/2 ciklusa po elementu; ako bi se petlja odmotala 8 puta, onda bi imali 1/4 ciklusa po elementu
      - povećana veličina koda
        - može dovesti do većeg broja promašaja u kešu (zašto?)
      - raste kompleksnost kompajlera
        - za veća tela petlji može ponestati registara za držanje svih neophodnih vrednosti, pa su potrebne kompleksne analize koda

- Paralelizam unutar procesora
  - Korišćenje ILP u kompajlerima
    - smanjenje uticaja skokova
      - kod petlji, loop-unrolling rešava jedan deo
      - za skokove se može koristiti dinamičko predviđanje
        - branch-prediction buffer (branch history table)
          - mali bafer koji se indeksira najnižim bitima adrese naredbe, sadrži 2 bita
          - najviši bit kaže da li je prošli put predviđanje bilo dobro
          - potrebna su dva suprotna dešavanja da se stanje izmeni





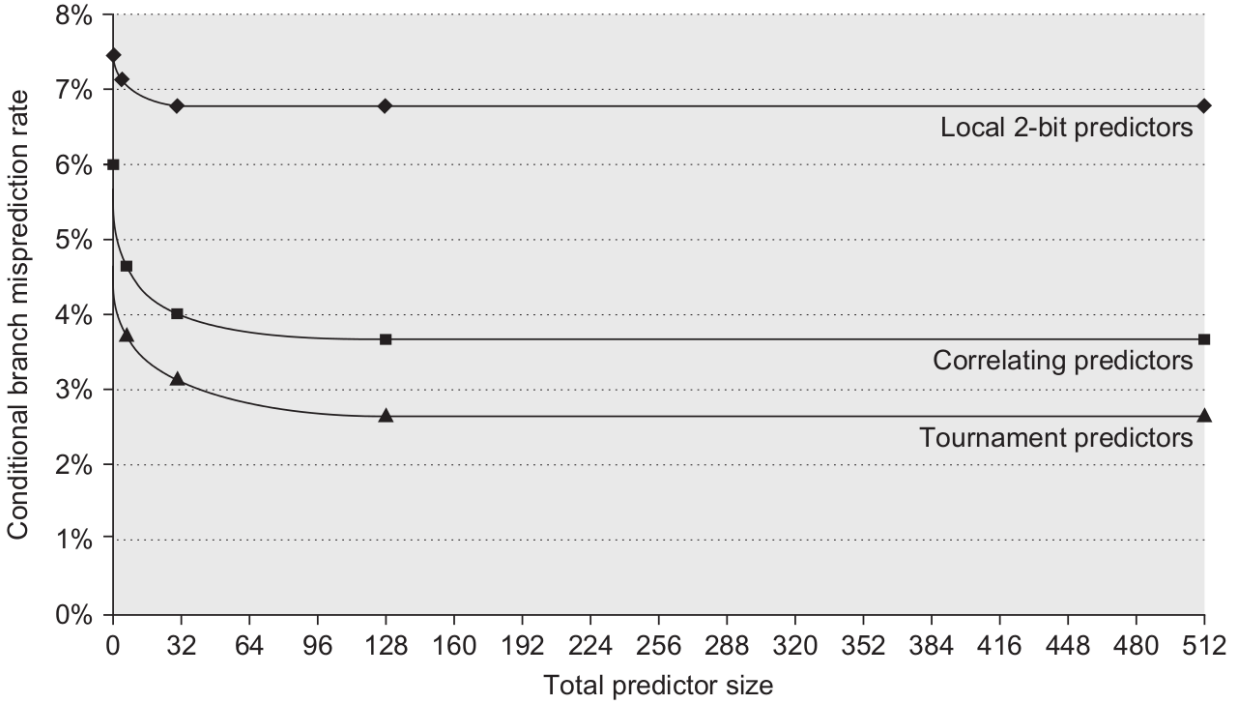
- Paralelizam unutar procesora
  - Korišćenje ILP u kompajlerima
    - smanjenje uticaja skokova
      - raniji skokovi mogu uticati na tekući - correlating predictor (two-level predictor)
        - može se napraviti bafer za predviđanje sa po dva elementa po skoku, gde bi se birao prvi ili drugi u zavisnosti od rezultata prethodnog skoka; može se napraviti i sa uračunavanjem više prethodnih skokova (za n prethodnih skokova, za svaki skok treba 2^n elemenata u tabeli)

```
if (aa==2)
    aa=0;
if (bb==2)
    bb=0;
if (aa!=bb) {
    ....
```

```
    addi x3,x1,–2
    bnez x3,L1      skok b1 (aa!=2)
    add  x1,x0,x0   aa=0
L1: addi x3,x2,–2
    bnez x3,L2      skok b2 (bb!=2)
    add  x2,x0,x0   bb=0
L2: sub  x3,x1,x2   x3=aa-bb
    beqz x3,L3      skok b3 (aa==bb)
    ...
```

- Paralelizam unutar procesora
  - Korišćenje ILP u kompajlerima
    - smanjenje uticaja skokova
      - turnirsko predviđanje - tournament predictors:

- nekada je bolje posmatrati samo ponašanje skoka bez uticaja prethodnih (local predictor), a nekada je bolje uključiti i analizu prethodnih (global predictor)
- odradi se i lokalno i globalno predviđanje za svaki skok
- doda se jedan viši nivo koji bira globalno ili lokalno predviđanje, u zavisnosti od toga koje je poslednji put bilo tačnije (opet korišćenjem 2 bita za beleženje, pa su potrebna dva uzastopna ista događaja da se promeni stanje)
- bolje prilagođenje različitim skokovima



- Paralelizam unutar procesora
  - Korišćenje ILP u kompajlerima
    - smanjenje uticaja skokova
      - dalja poboljšanja predviđanja
        - kombinovanje više predviđanja, gde svako predviđanje koristi različit broj prethodnih predviđanja (uključujući i 0); bira se ono koje trenutno daje najveću tačnost za zadati skok
        - uvođenje tagova za tokove izvršavanja skokova
        - uvođenje bita validnosti za elemente tabele (ubrzava inicijalno punjenje - neće se gledati nevalidni elementi)
  - Dinamička (hardverska) preraspodela naredbi (dynamic scheduling)
    - nema potrebe za rekompajliranjem (većina komercijalnog softvera se distribuira u binarnom obliku)
    - bolja preraspodela za scenarije gde se ne može statički odrediti najbolji redosled
    - omogućava procesoru da se nosi sa nepredviđenim kašnjenima

- Paralelizam unutar procesora
  - Dinamička preraspodela naredbi
    - bez dinamičke preraspodele, ako jednoj naredbi treba rezultat prethodne, ona mora čekati, kao i sve ono što sledi posle nje
    - primer

```
fdiv.d f0,f2,f4    f0 <- f2/f4
fadd.d f10,f0,f8   f10 <- f0+f8
fsub.d f12,f8,f14  f12 <- f8-f14
```

      - fadd zavisi od fdiv, ali fsub ne zavisi ni od jedne
      - fsub bi se moglo izvršiti u paraleli, ako bi se zaključilo da je nezavisna
        - naredba se ubacuje u pipeline
        - izvršava se čim joj operandi postanu dostupni (iako u source kodu to nije tako - out-of-order execution), što dovodi i do ranijeg dobijanja rezultata (out-of-order completion)

- Paralelizam unutar procesora
  - Dinamička preraspodela naredbi
    - izvršavanje van redosleda može dovesti do WAW i WAR opasnosti po podatke
    - primer

```
fdiv.d f0,f2,f4    f0 <- f2/f4
fmul.d f6,f0,f8    f6 <- f0*f8
fadd.d f0,f10,f14  f0 <- f10+f14
```

      - antizavisnost između fmul i fadd; ako se fadd krene izvršavati pre nego fmul završi, postoji WAR opasnost (šta je to ovde tačno?)
    - izvršavanje van redosleda uvodi komplikacije sa izuzecima
      - moraju i smeju se desiti samo oni izuzeci koji bi se desili kada bi se kod izvršavao sekvencijalno
        - čuvanje i odlaganje izuzetka
      - mogu se desiti neprecizni izuzeci (imprecise exceptions)
        - procesor može biti u stanju koje nije potpuno identično stanju kada se naredbe izvršavaju sekvencijalno
        - naredbe iza one koja je izazvala izuzetak mogu biti već izvršene
        - naredbe pre one koja je izazvala izuzetak se možda još nisu završile

- Paralelizam unutar procesora
  - Dinamička preraspodela naredbi
    - faze u petostepenom pipeline-u:
      - IF - instruction fetch, ID - instruction decode, EX - execution, MEM - memory access, WB - write-back
    - da bi se dinamička preraspodela mogla odraditi potrebno je da faza dekodiranja naredbe bude iz dva dela:
      - issue - dekodiranje naredbe
      - read operands - čekanje da prestanu sve opasnosti po podatke, pa čitanje operanada
      - početak i kraj izvršavanja naredbe obeležavaju period kada je ona “u izvršavanju”, i upravo preklapanje ovih faza raznih naredbi je srž dinamičke preraspodele; ovo zahteva i više funkcionalnih jedinica (zaduženih za izvršavanje)
    - sve naredbe prolaze kroz fazu dobavljanja u redosledu u programu, a već od faze čitanja operanada se mogu desiti zastoji i izvršavanja unapred
    - jedan od prvih algoritama za odabir naredbi za izvršavanje unapred, Scoreboarding, je nastao na CDC 6600 (1964)