マイクロプロセッサ・アーキテクチャ
- コンピュータの中核部分を考える

東京大学 大学院情報理工学系研究科
入江 英嗣

2018/11/12

自己紹介

http://www.mtl.t.u-tokyo.ac.jp/~irie

- 所属
  - 電子情報工学/電子情報学専攻
- 研究分野
  - コンピュータシステム
    - **コンピュータ・アーキテクチャ**
    - **ヒューマン・コンピュータ・インタラクション**
    - **ディペンダビリティ/セキュリティ**
- 「**アドバンスト・コンピュータアーキテクチャ**」
  - 来年度開講

今日のお話

- **マイクロプロセッサ・アーキテクチャ**の動向
  - 現在の主なプロセッサ設計戦略
  - 近年のCPUアーキテクチャに求められる役割
- **CPUアーキテクチャ技術**
  - RISC向けマイクロアーキテクチャ技術概観
- **研究紹介：STRAIGHTアーキテクチャ**
  - 高性能高効率かつ設計簡略化を実現

マイクロプロセッサ・アーキテクチャの動向

2018/11/12

プロセッサとは？

- メモリの指示内容に従ってメモリを書き換えるユニット
  - ノイマンの言うところのCC, CA (残りはI,O,M,R)
    - 元祖CPU
  - （一般的には）プログラムを実行できる回路
    - GPU, DSP, TPU，とりわけ（SoC内のホストアーキテクチャという意味での）CPU
- マイクロプロセッサ
  - 一つのICに収まっているプロセッサ
  - 一つのICに複数収まるようになるとそれぞれをコアと呼んでその集合体をプロセッサと呼ぶように
    - 過渡期にはチップマルチプロセッサと呼称

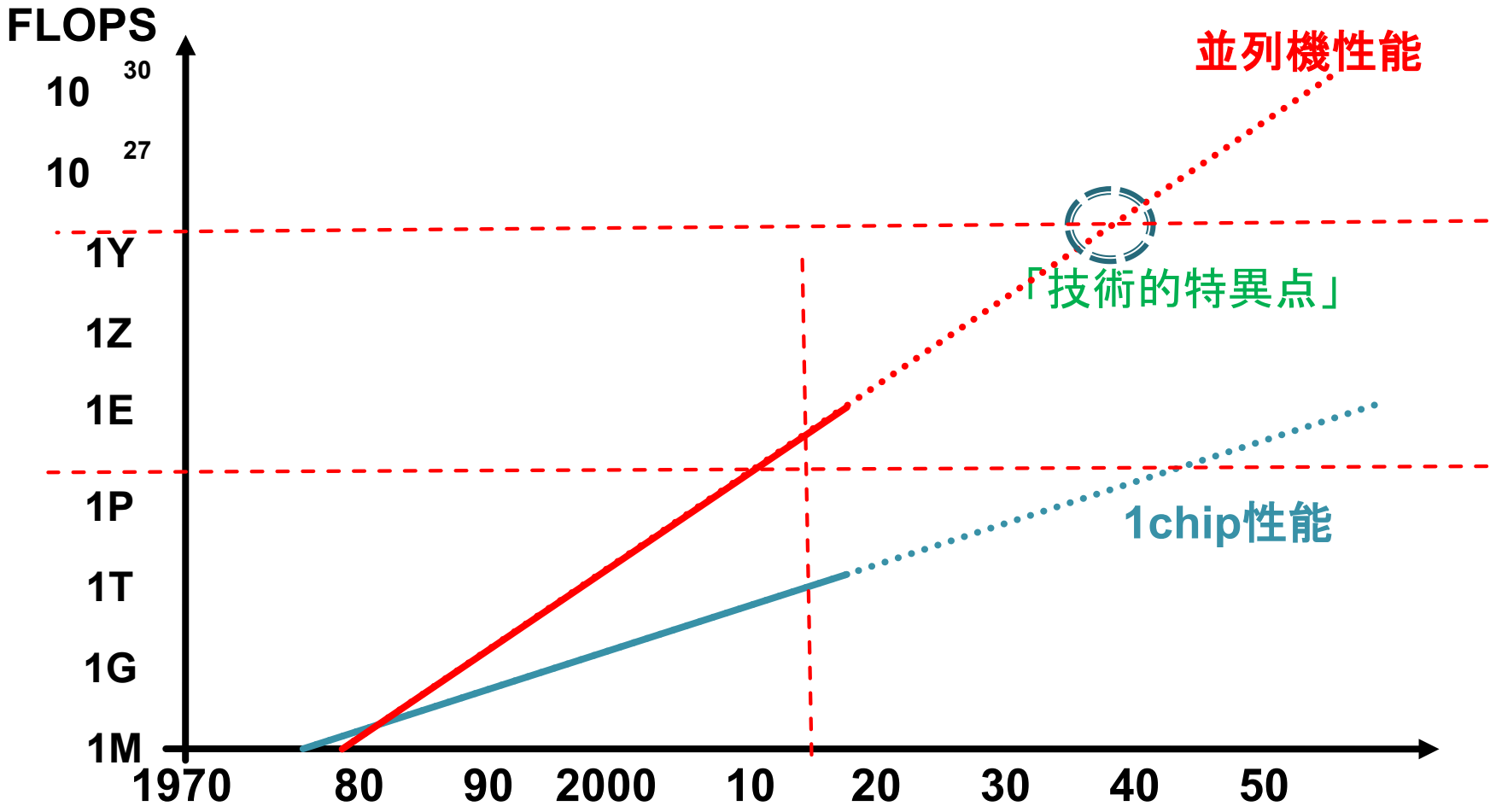
プロセッサ性能向上は生活スタイル一変のトリガ

- アプリケーション/アルゴリズムの必要計算力・許容コスト
  - プロセッサ性能がこれを越えれば爆発的に浸透
  - 各用途でそれぞれ異なるバランス

IoT

携帯端末	**パソコン**	**サーバ**	**スパコン**
バッテリー容量に対する計算力	**1チップのコスト＆計算力**	**コスト・電力あたりのプロセッサ数**	**電力に対する計算力＆通信能力**

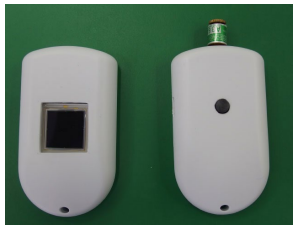
性能面


FLOPS
10 30
10 27
1Y
1Z
1E
1P
1T
1G
1M
1970
80
90
2000
10
20
30
40
50
並列機性能
「技術的特異点」
1chip性能

フォームファクタ面

- 従来のイメージから変化
- あらゆる物が急速にコンピュータ化
  - 従来は「**コントローラ**」
- 背景：**プロセッサ**の進歩
  コンピューティング能力付与のコスト低下
  - **1mm²**，**100mW**あれば
    それなりの**CPU**が**1GHz**で動く
  - **10W**も用意すれば
    マルチコアと**ベクタユニット**がつく
- **CPUが入ってくる意味**：プログラマブル
  - **ユーザの突発的な創意**をプログラムにして
    様々な要求に応えることができる

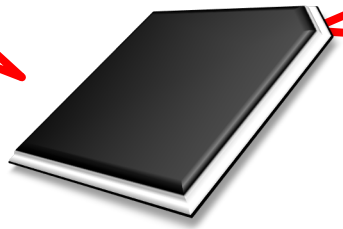
身の回りのあらゆるものがプログラムを理解



「良い」プロセッサとは？

計算能力
IPS　FLOPS
高いほど良い

コスト
数がでるなら‥**面積**
でないなら‥**開発費**
（プロセス世代）
安いほど良い



消費電力
PD/ED積　TDP
低いほど良い

信頼性
FIT　寿命
セキュリティ機能
なければ使えない

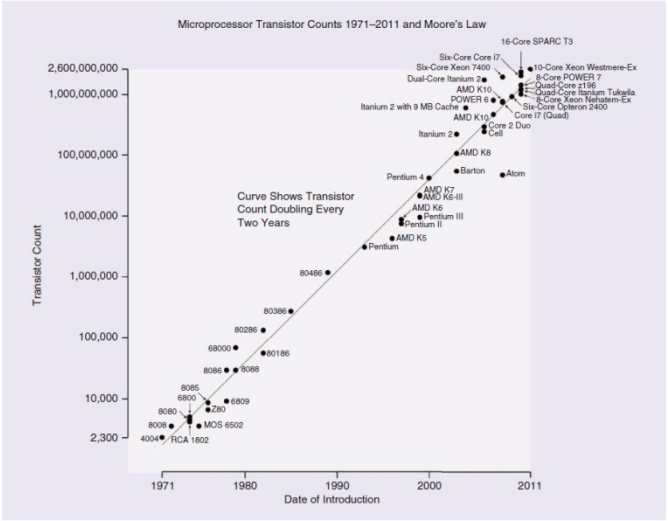
使いやすさ
互換性、プログラマビリティ、ミドルウェア整備・・・
≒**ユーザ数**
多いほど良い

寡占化が進みやすかった⇒ 近年，変化の傾向

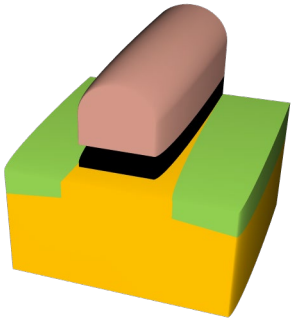
継続的なプロセッサ成長の源泉：半導体微細化

- **ムーアの法則**
  - **「半導体の集積度は3年で4倍になる」**
  - 正確には**経験則/経営目標**

- 使える**リソース量の指数的増加**
- **Dennardスケーリング**
  - CMOSの嬉しい特性
  - 素子が小さくなると性能・効率共に向上



[M.Guarnieri, 2015]



スケーリングファクター*K*:

デバイスの寸法 *1/k*

電圧 ***1/k***

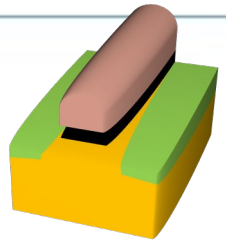
遅延 ***1/k***

電力消費/デバイス ***$1/k^2$***

電力消費/面積 ***1***

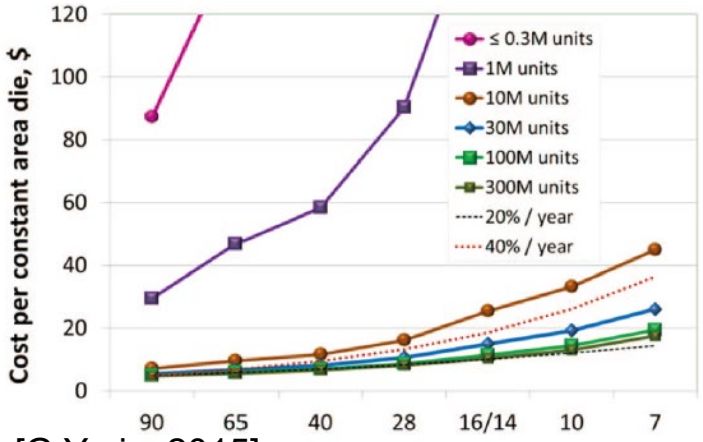
CMOS微細化進展と共に出てきた課題達

- **メモリ・ウォール**
  - チップ性能に見合ったキャッシュやバッファが必要
- **配線遅延**
  - RAMのポート数，容量と速度の両立が不可
  - チップ内の地理的影響
- **ホット・スポット**
  - 周波数の制限
- **リーク電力**
  - デナードスケーリングの破綻
- **プロセスばらつき**
  - 速度，電力面でのゲイン減少
- **ダーク・シリコン**
  - 同時Tr稼働数に制限
- **デザイン・製造ハザード**
  - 高価な最新プロセスに見合った効果が必要



デバイスの寸法 *1/k*

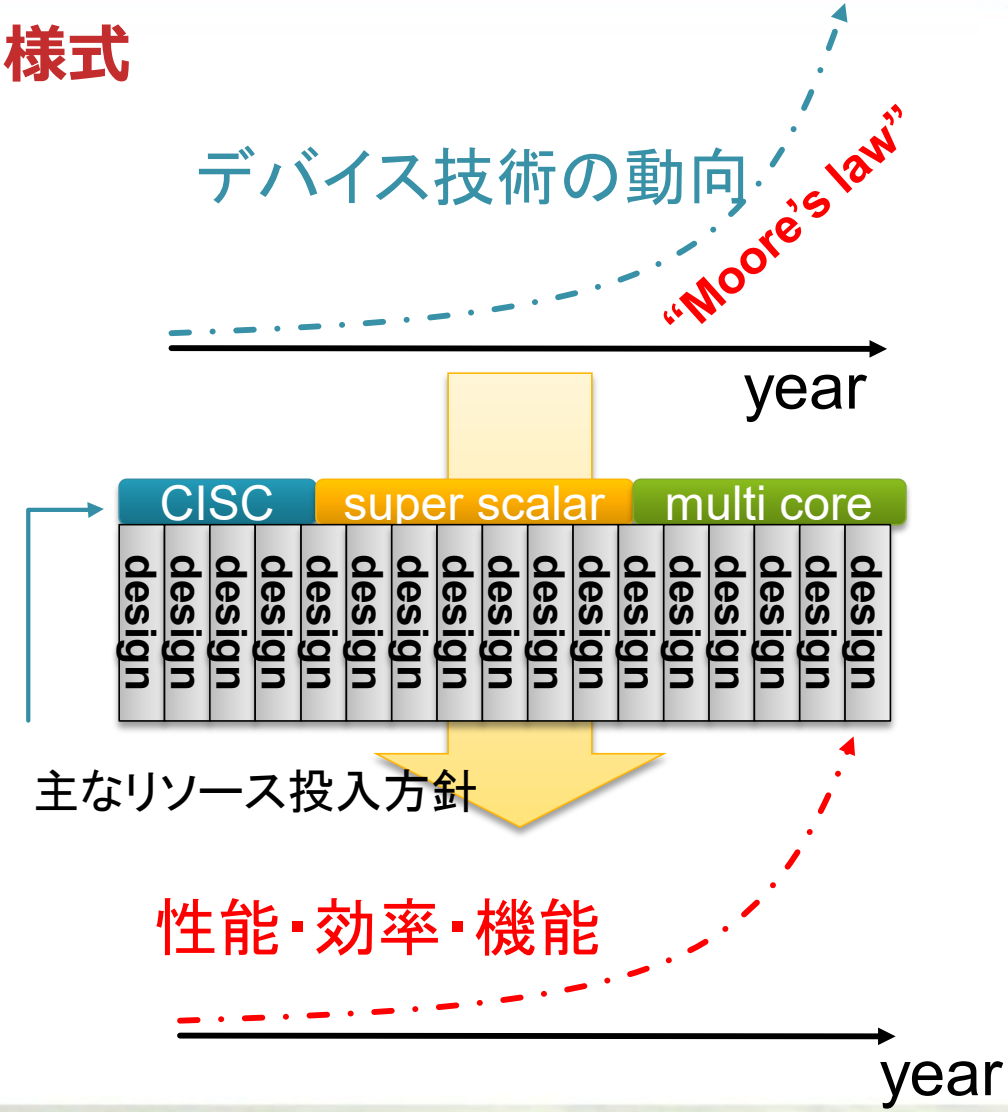
電圧	***1/k*** ⬅
遅延	***1/k***
電力消費/デバイス	***$1/k^2$***
電力消費/面積	***1***



[G.Yeric, 2015]

計算機「アーキテクチャ」による性能向上

- コンピュータの**構成法・様式**
  - コンピュータとは「**このように作るもの**」
  - プログラマ・OSから見た**コンピュータ仕様**
  - トランジスタ**リソースの使い方**の指針
    - 量を活かす
    - ワークロードへの合致のさせ方

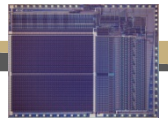


アーキテクチャの転換期と拡張期

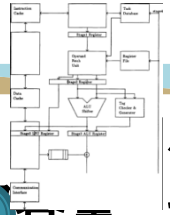
http://www5.pcmag.com/media/images/9183-power4-die.jpg
http://www.cs.berkeley.edu/~pattrsn/Arch/RISC1.jpg
MASA: Halstead+, 1988
Alpha 21264 Kessler, 1999
Broadwell-Y: MPR 9/15/14

CISC

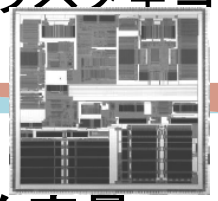
1980 RISC



1990 MT



2000 クラスタ型コア



命令の種類

パイプライン規模

パイプライン規模

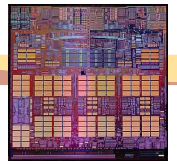
容量

リソース利用効率の低下

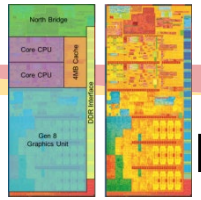
ILP収穫逓減

配線遅延

2005 Multicore



2010 Hetero. MC



?

クラスタ数

コア数

DLPコア数
コアの種類

?

コアの消費電力に対する収穫逓減

パッケージ消費電力の限界/TLP収穫逓減

“Pollack’s Rule”

“Dennard Scalingの破綻”, “Dark Silicon”

近年のアーキテクチャ技術傾向

- **パイプライニングによる周波数上昇**の効果は飽和
  - 電力の制限
  - メモリのレイテンシ
- **並列性の利用，処理効率の向上**
  - なるべく多くの処理を同時に行う
  - 同じ仕事なら少ない処理で行う
- **メモリ階層の効率化**
  - キャッシュ・マネジメント
- **近似計算**
  - 実用上問題ない範囲で精度を落として速度と効率を向上

ILP TLP DLP

- **Instruction Level Parallelism**
  - CPUコア一つあたりにリソースをかけることで抽出
  - スレッド実行そのものが高速化
- **Thread Level Parallelism**
  - コア数を増やすことで抽出
  - 複数スレッドで構成されたタスク実行が高速化
- **Data Level Parallelism**
  - SIMD, SIMTユニットの実行幅を増やすことで抽出
  - 流儀に添って書かれたプログラム部分の実行が高速化

Pollack's Rule あるいは√Trの法則

- プロセッサ成長に関する経験則
  - （ある方法論で得られる）
    性能はつぎ込んだリソースの平方根に比例する
    - 価格，トランジスタ，電力
- スーパリニア→リニア→飽和
  - リニア帯までで組み合わせた方がリソース効率は良い
    - **ILP** vs. **多ポートRAMの指数的コスト**
    - **TLP** vs. **Amdahl's Law**
    - **DLP** vs. **アルゴリズムの出現頻度**

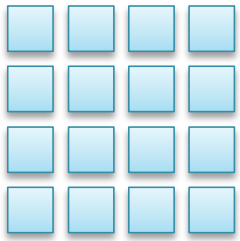
コアの種類と特性

	プログラムの書きやすさ	得意ワークロードの多さ	電力効率	性能
CPU (big)	◎	◎	×	○リソース次第
CPU (little)	◎	△	○	×
GPU	△	△	○	◎
DSP	○	△	◎	○
FPGA	×	○	◎	○
ASIC	ー	×	◎	◎

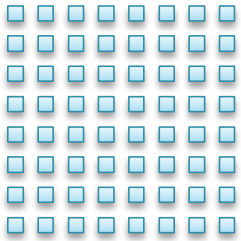
- 一つのプロセッサパッケージ内に複数個・複数種類のコアを実装

Pollackの法則の帰結： トレードオフの重層化

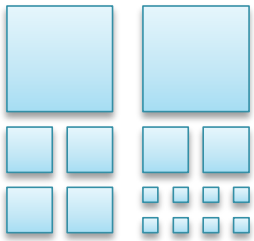
- できることは取り込まれていく
  - 「マルチコアかアクセラレータか？」
    ⇒リソースに合わせて**両方使う**，が基本
    - かつての「SMT vs. CMP」の議論同様



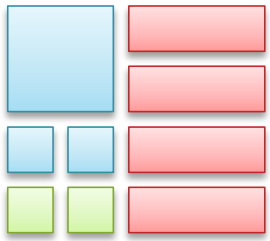
マルチコア



メニーコア



ビッグ・リトル
(ヘテロジニアス・マルチコア)



ヘテロジニアス・ISA

ヘテロジーニアスISAマルチコアの利点

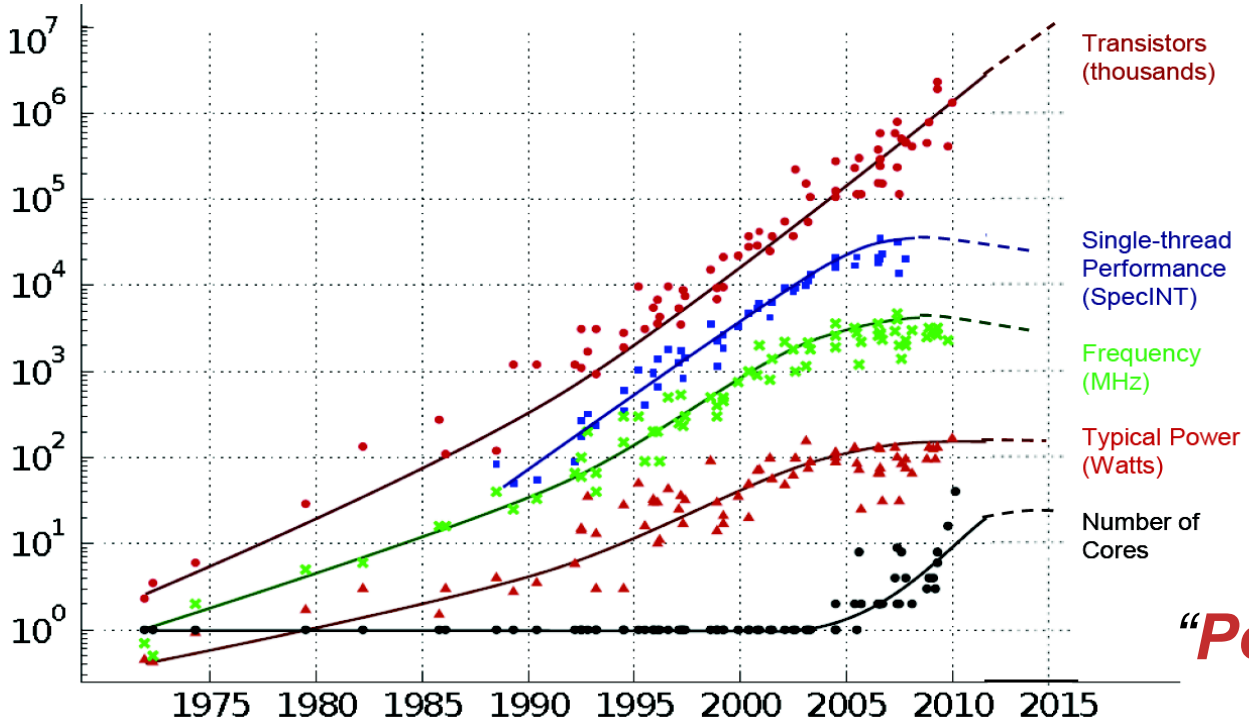
- タスクに合わせて適したコアが動く
  - 「コア使用率が犠牲となるが
    いずれにしろダークシリコンなのでOK」
- 最低限の実行はCPUが保証
  - ISAや汎用性の制約なく新方式を導入可能
- ディペンダビリティ，セキュリティ

- ただしワンオフ，**成長戦略ではない**
  - コア種類を増やし続けてもすぐに飽和
  - 目的・時期にあった素早い設計が鍵

現在のプロセッサの成長戦略

- 地道な**コア数増，シングルスレッド処理能力増**
  - （一部の限られたプレーヤのみ）
  - 緩やかに伸び続けている
- 用途に合わせたプロセッサ設計
  - コアの**プルーニング**
  - 目的にそったコア構成（**テーラリング**）
    - **ワークロードの特徴**，**並列性**，**汎用性**があるほど特に有利
    - Deep Learning/ セキュリティHW：この傾向に合致
  - **Cache Coherency Network**
    - ヘテロ構成のコア間の連携強化

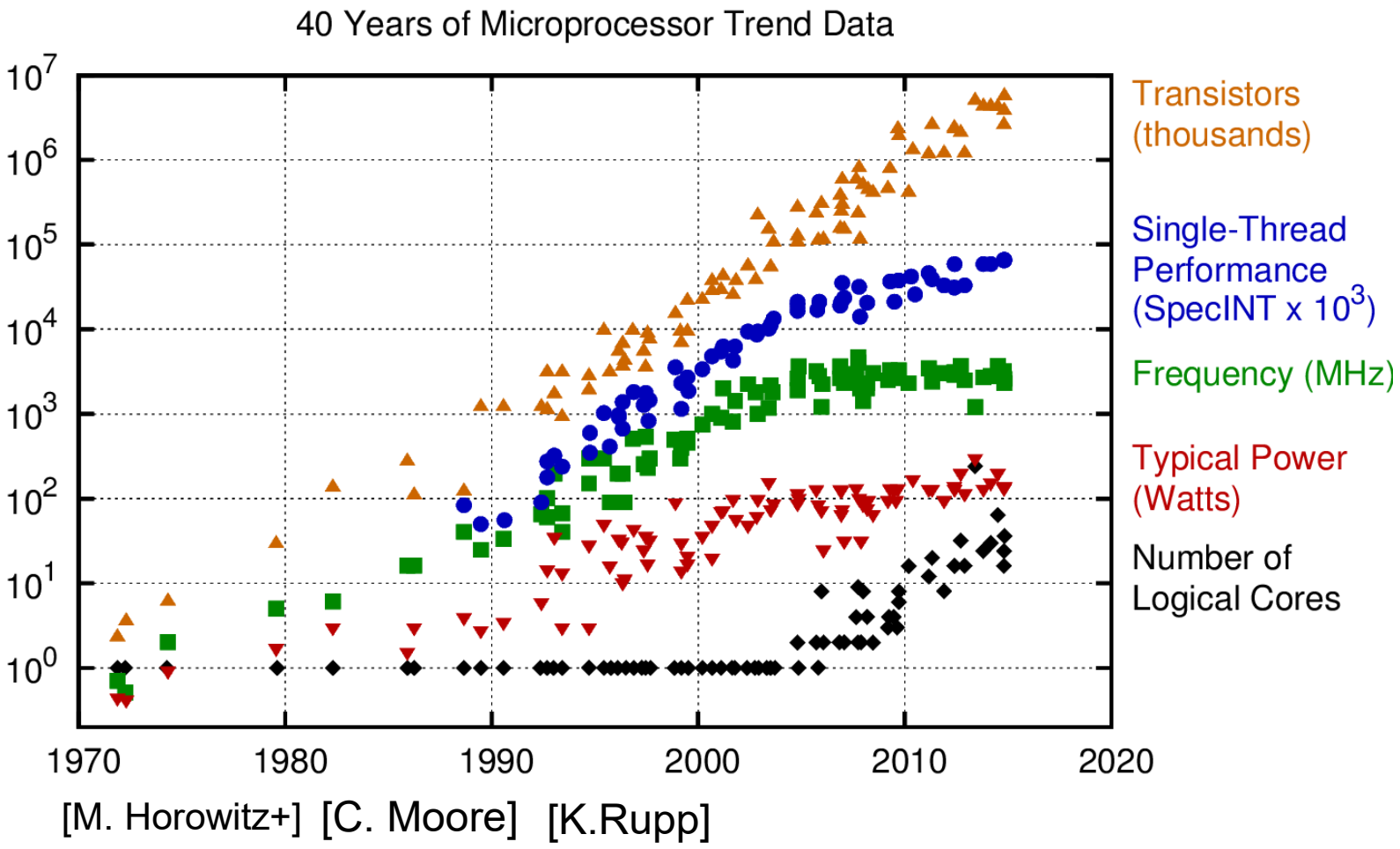
プロセッサ成長の傾向予測



[M. Horowitz+] [C. Moore]

"Post Moore"

素子の進歩が止まるなら
カスタムプロセッサが有利
様々な見通しが
後退しているのは確か


40 Years of Microprocessor Trend Data
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1970
1980
1990
2000
2010
2020
Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores
[M. Horowitz+] [C. Moore] [K.Rupp]

汎用プロセッサ vs. アクセラレータ

- 汎用プロセッサ(CPU)
  - プログラム開発が容易
  - 新ハード登場時：
    ソースまたはバイナリレベルでSW資産再利用可能
  - ユーザ多数　⇒　最新プロセスで開発可能
- アクセラレータ
  - 桁の違う効率を達成可能
  - 新ハード登場時にはSW書き直し（一部または全体）
    場合によってはプログラマが手法再学習
  - ユーザ数に応じて適切なプロセスで開発

- プロセスの成長速度がバランスポイントを決定

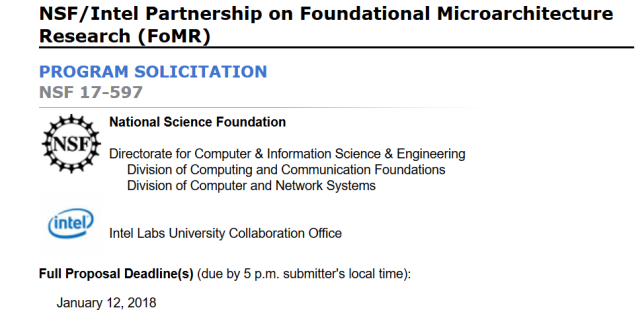
象徴的なムーブメント2つ

- NSFプロジェクト
  - シングルスレッド技術の革新をターゲット
    - 新たなILP技術がCPUに必要
- RISC-Vのブーム
  - オープンコアの急激な需要増

NSF 17-597 (FoMR)

- **IPC向上**をターゲットとしたグラント
  - 「**斬新なマイクロアーキテクチャ**」を支援
  - 50万$規模 x 6件

1. マイクロアーキテクチャから
   コード生成までを含むIPC向上技術
2. チップ全体のリソースを使って
   シーケンシャル実行を加速するターボ技術
3. IPC向上に効果的なコンパイラコード生成技術



NSF/Intel Partnership on Foundational Microarchitecture Research (FoMR)

PROGRAM SOLICITATION
NSF 17-597

National Science Foundation

Directorate for Computer & Information Science & Engineering
Division of Computing and Communication Foundations
Division of Computer and Network Systems

Intel Labs University Collaboration Office

Full Proposal Deadline(s) (due by 5 p.m. submitter's local time):

January 12, 2018

https://www.nsf.gov/pubs/2017/nsf17597/nsf17597.htm

- シングルスレッド性能の新方式に投資が始まる(2018.1応募〆)

RISC-V

- UCB発のオープンアーキテクチャ
  - 最初はコース課題
- 最新のRISCアーキテクチャ
  - 技術的にはここまでの知見を反映
    - レジスタウィンドウなし，遅延スロットなし
    - 圧縮命令，ベクタ拡張
- どちらかといえば運営の提案
  - 命令セットをオープンかつ固定にしてハブ化
    - 命令セット不変を可能とするためのモジュール化
  - ある種，CPUアーキテクチャ成長に対する見切り
    - 100年後も使える最後のRISC，CPU

カスタムチップのモチベーション

- DL, ブロックチェーンなど特定用途に計算需要
  - クラウドだけでなくエッジ用途の必要性大
- ハードウェアインジェクションなどセキュリティ上の脅威

- 自前のアクセラレータ製造需要
  - アクセラレータであれば性能向上および製造は比較的容易
  - ホストとなるCPUの需要
    - 高性能CPUはライセンスが高額

    ⇒オープンCPUアーキテクチャの注目度上昇

高性能新CPUアーキテクチャの「機」

- 高性能・高効率のためならヘテロISAが受容
  - 少し前までは互換性制約が絶対
  - コンパイラフレームワークの発展：LLVM
- カスタムチップの商用価値の拡大，
  そのためのホストCPUの需要
  - ソフトコアでは不足
  - ライセンスコアは幅広いプレーヤからは敬遠
  - RISC-V: 従来アーキテクチャと同様に性能・効率は制約
- もしこのタイミングでRISCを上回る性能で，設計容易，
  オープンなCPUアーキテクチャが登場すれば･･･
  - 新規プレーヤでも参入できる可能性大

CPUマイクロアーキテクチャ技術

2018/11/12

最近の（汎用）プロセッサ

	デスクトップ Skylake(x64)	サーバ Haswell-EX(x64)	モバイル Cortex A73(ARM v8)	IoT Cortex A35(ARMv8)
プロセス世代	14nm	22nm	16nm	28nm
大きさ	数百mm^2	662mm^2	1.2mm^2	0.4mm^2
動作周波数	4GHz	2.5GHz	2.5GHz	1-2 GHz
消費電力	90W	数百W	0.8W	6mW
CPU構成	ベクタユニットつき8way oooSS x 4	ベクタユニットつき8way oooSS x 16	ベクタユニットつき8way oooSS x 4	1way 8stage
アクセラレータ	GPU	GPU	パッケージによる	パッケージによる
LLC	32MB L3	45MB L3	8MB L2	8KB-64KB L1 (1MB L2)

プロセッサ・アーキテクチャの汎用性と伸縮性

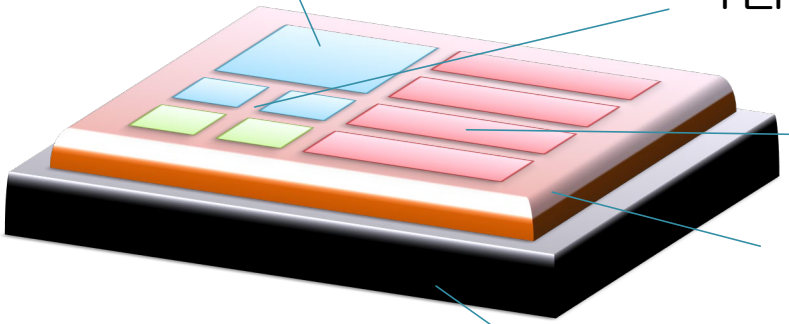
- **伸縮性**
  - 表の例だけでも
    面積で数千倍　W数で数十万倍
- **共通フレームワーク**
  - 核となるCPUアーキテクチャ
  - プロセス世代，命令セット
  - 構成する各要素
- **重層的な拡張**
  - CPUひとつあたりの規模
  - CPUの数
  - アクセラレータの種類，数
  - キャッシュ

Intel Skylakeのシングルスレッド性能成長戦略[Doweck+, 2017]

- 命令ウィンドウ拡張，実行幅は（おそらく）据え置き
  - スケジューラ 60 > **97**
  - ROB 192 > **224**
  - RF 168 > **180**
  - デコード幅 4 > **5-6**
- **予測技術**の改良・促進
- 演算器改良
- **パワーマネジメント**

プロセッサ成長の課題達

CPU(ILP): シーケンシャル部分の実行高速化効率化
トータルパフォーマンスに非常に重要



TLP: 同期オーバヘッドの軽減，インタコネクト，
キャッシュコヒーレンシ

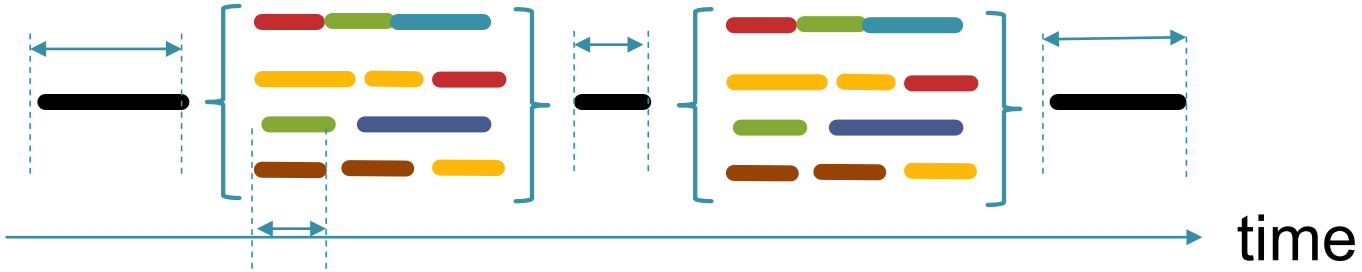
DLP: 柔軟性の獲得，アプリケーション特化

マイクロアーキテクチャ共通：
記憶階層，投機，学習と自己最適化，近似計算

新素材，製造技術，新計算原理

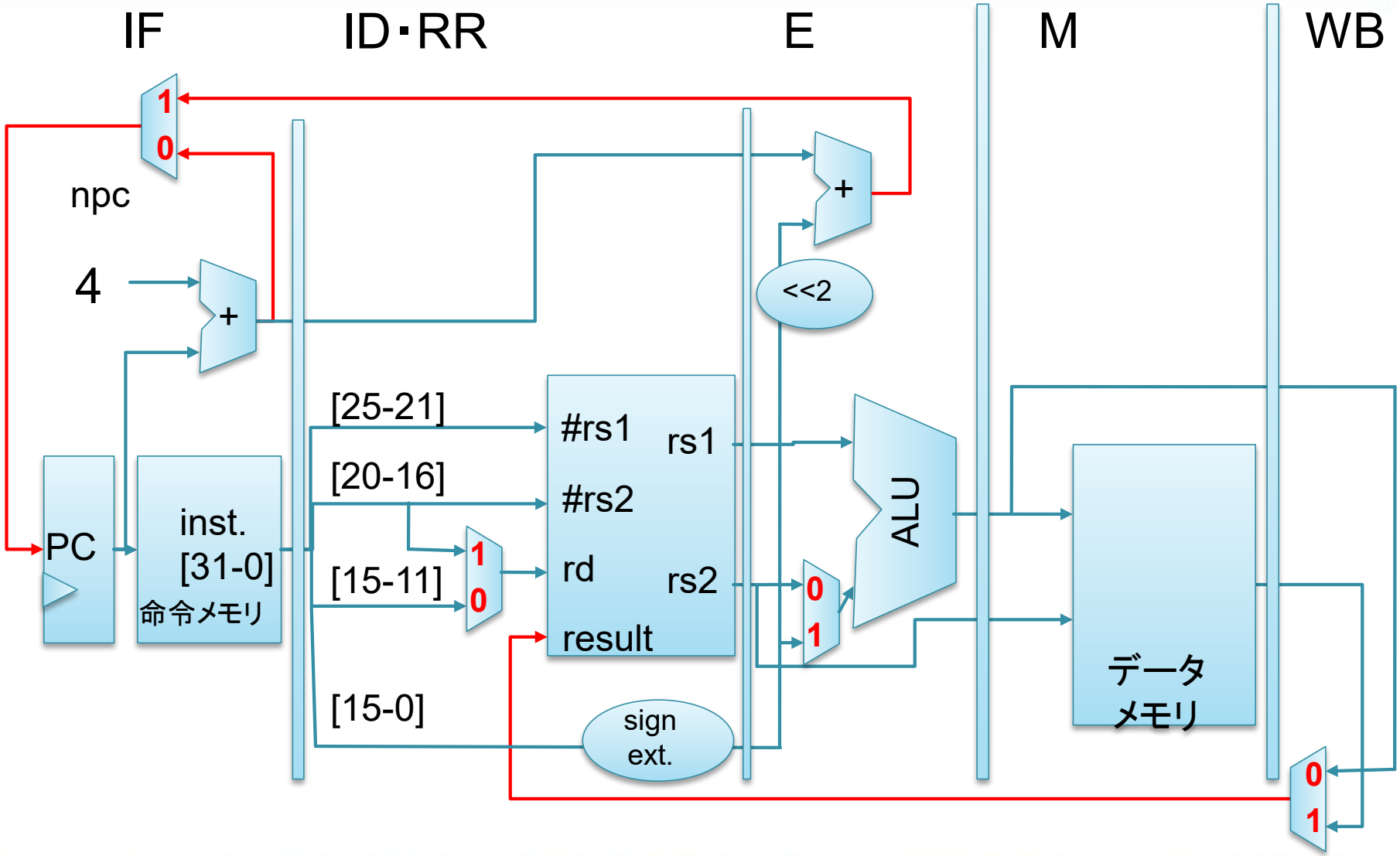
シングルスレッド性能の影響

- コア構成が豊かになるほど影響大
  - タスクの**クリティカルパス**となる**シーケンシャル部分**の性能に影響
  - DLP/TLP各スレッドの高速化にも影響

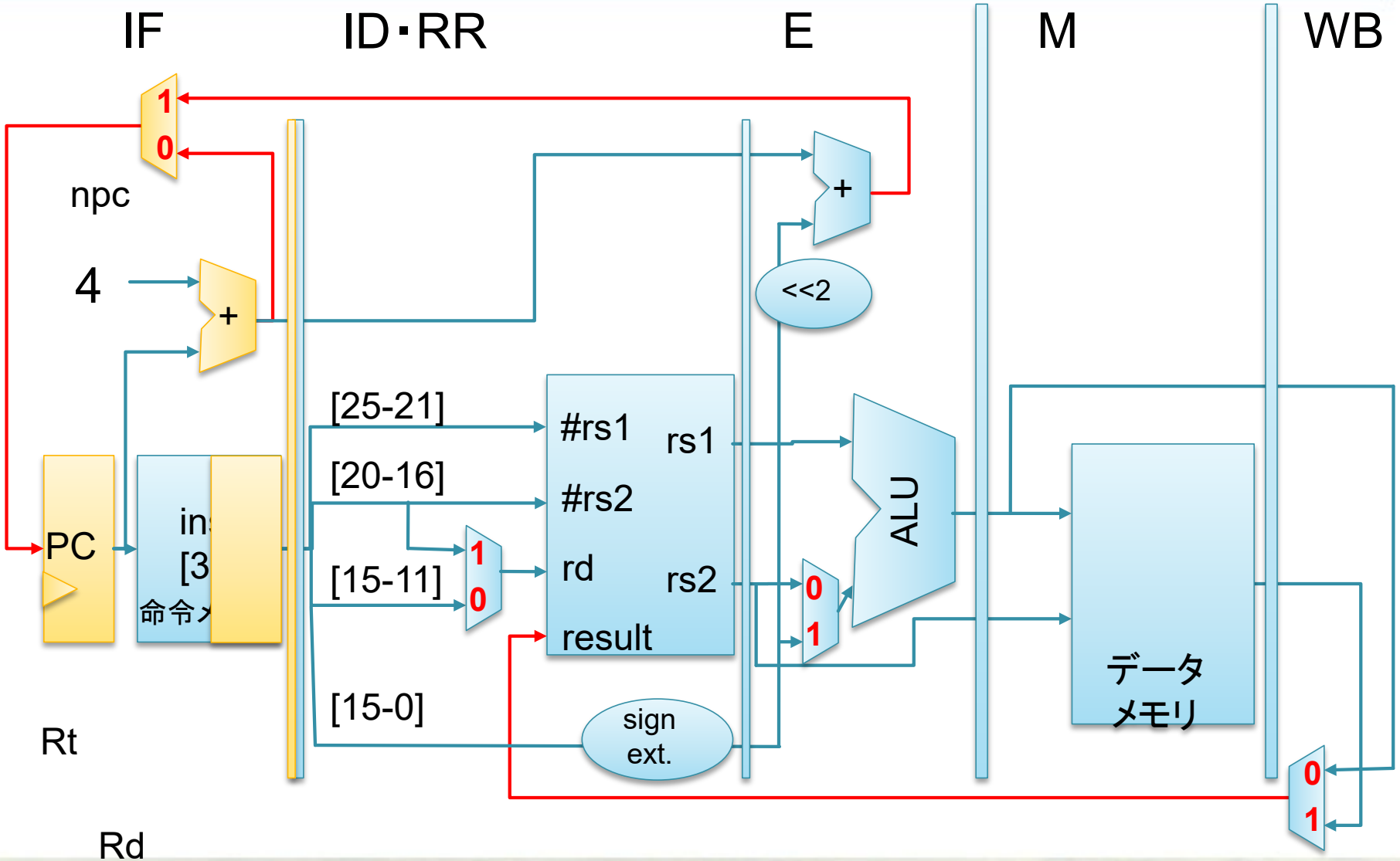


- 現状の高性能化の選択肢
  - **OoOスーパスカラ（ビッグコア）**のみ
  - **効率が悪**く，これ以上拡張しづらいことで知られるが**不可避的に実装**
    - ローエンドモバイルになると**贅沢品**

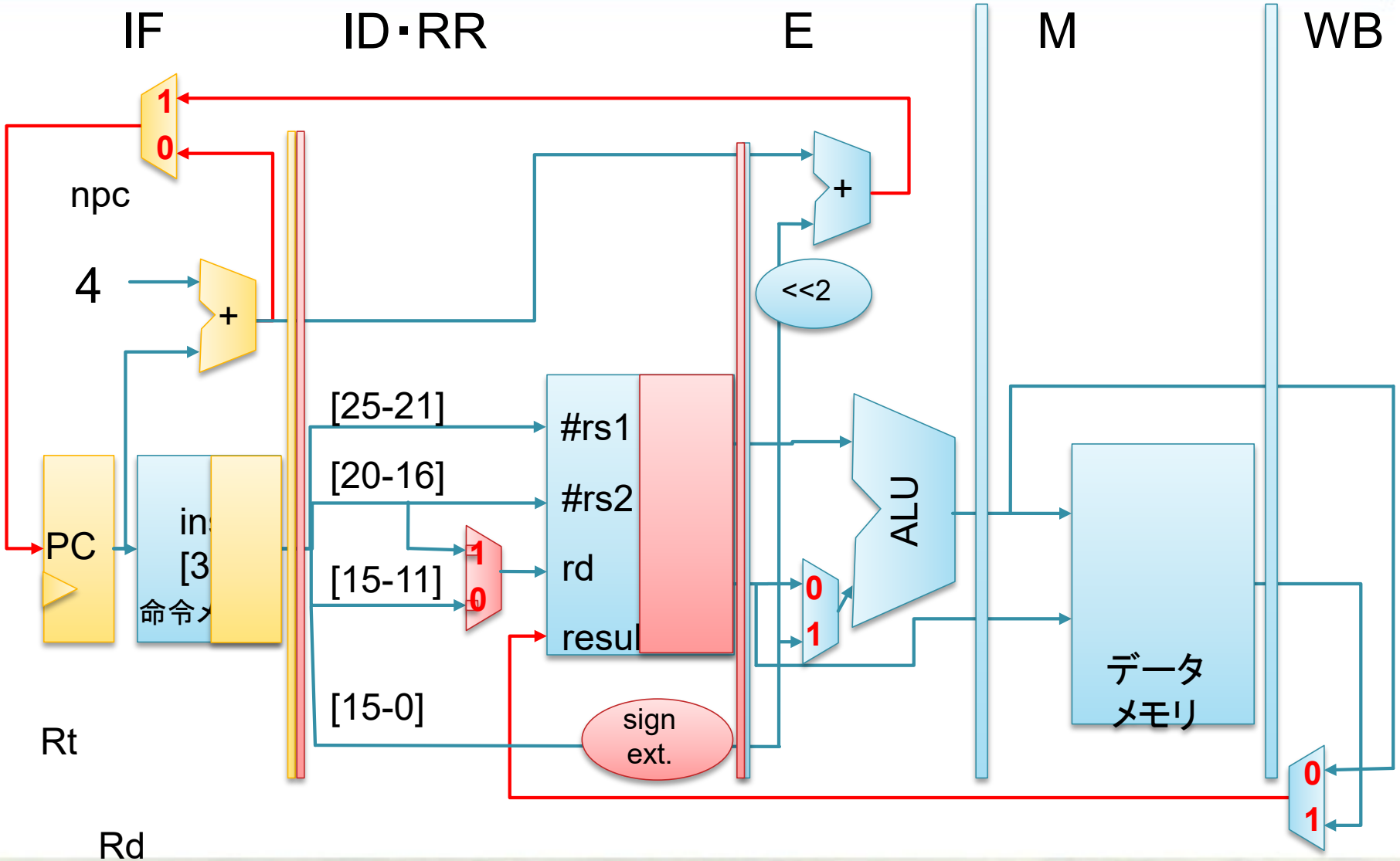
おさらい：スカラ・プロセッサ



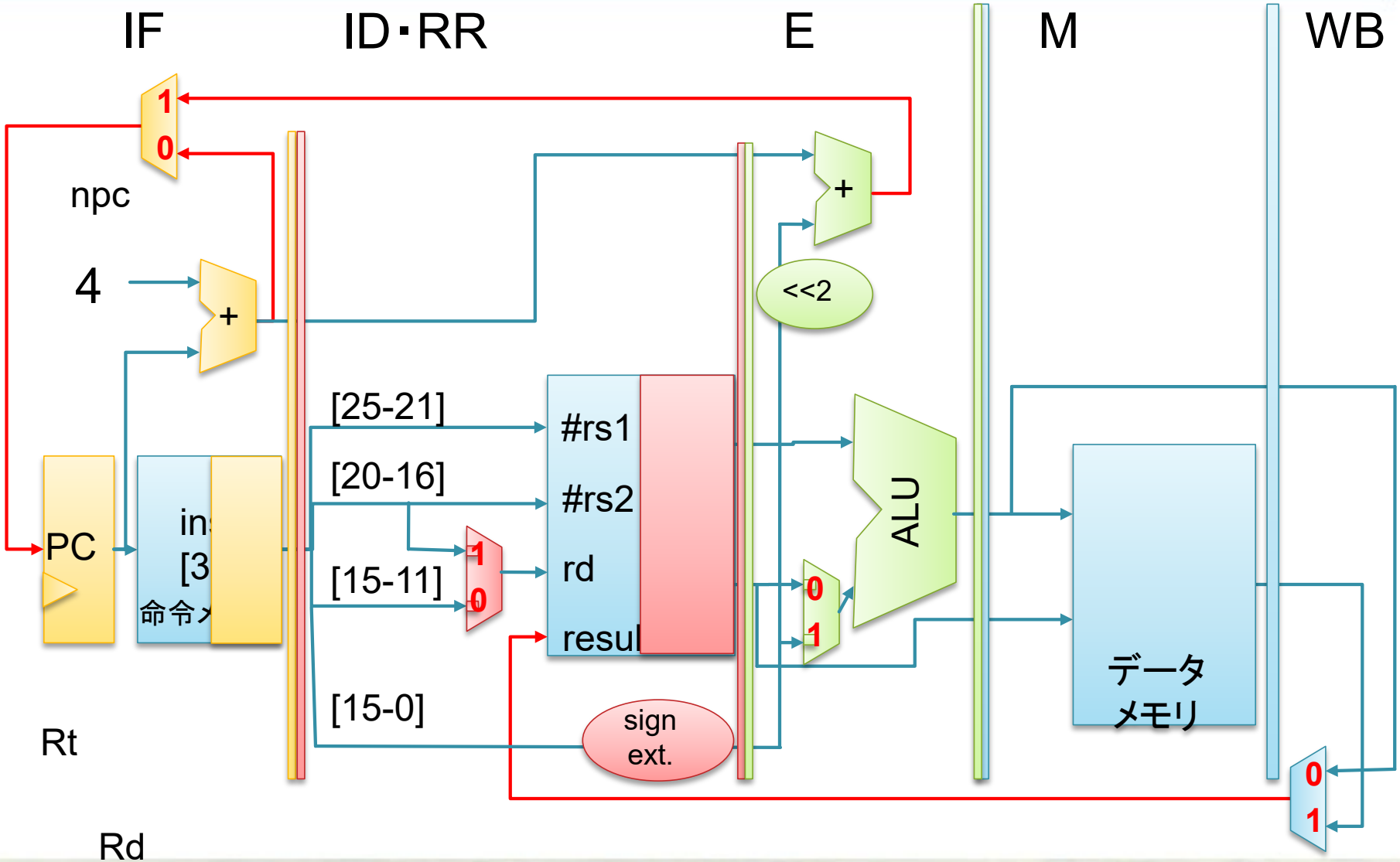
フェッチステージ


IF
ID・RR
E
M
WB
1
0
npc
4
+
PC
in
[3
命令メ
[25-21]
[20-16]
[15-11]
[15-0]
1
0
#rs1
rs1
#rs2
rd
rs2
result
sign
ext.
+
<<2
ALU
0
1
データ
メモリ
0
1
Rt
Rd

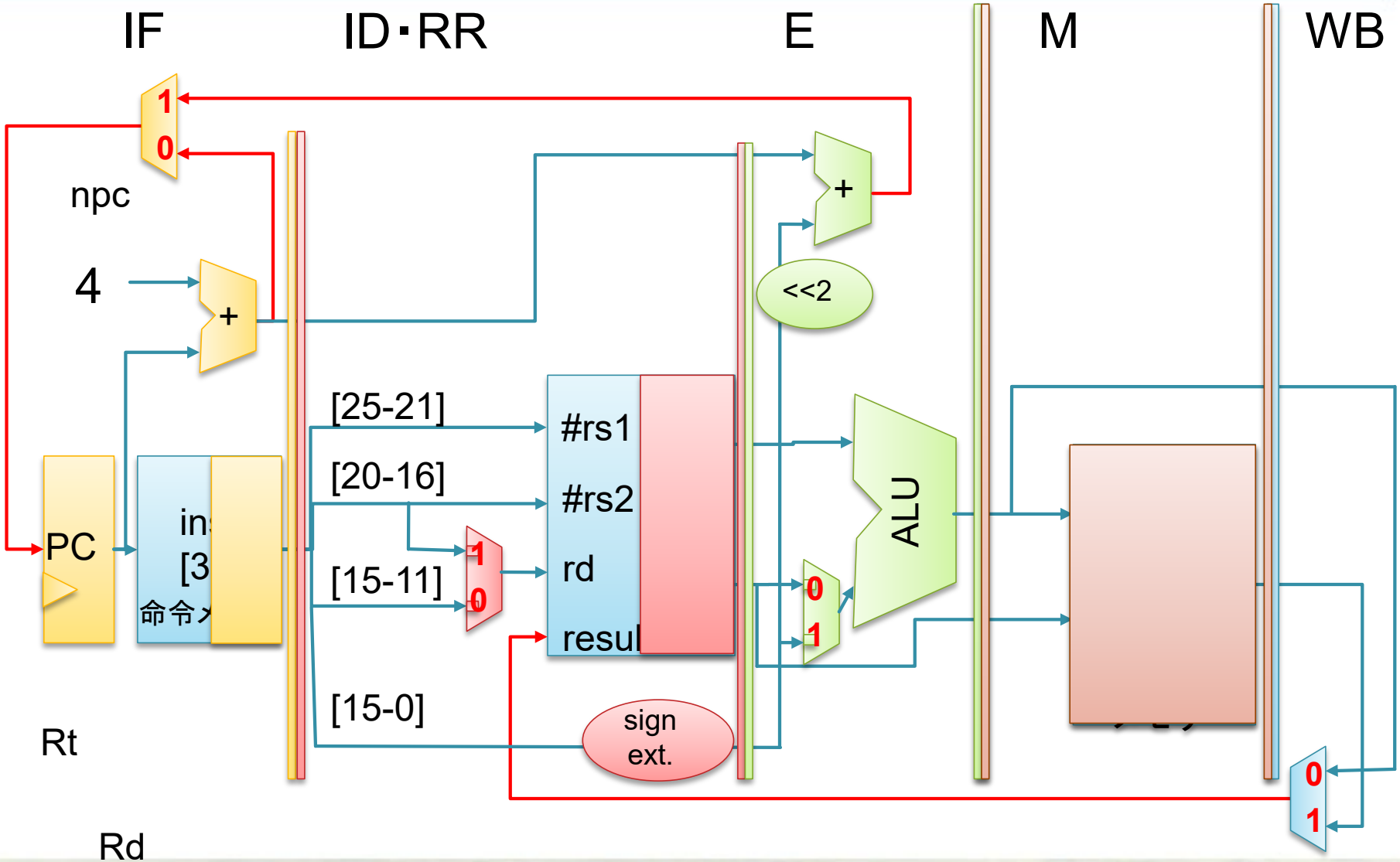
デコード・レジスタリードステージ



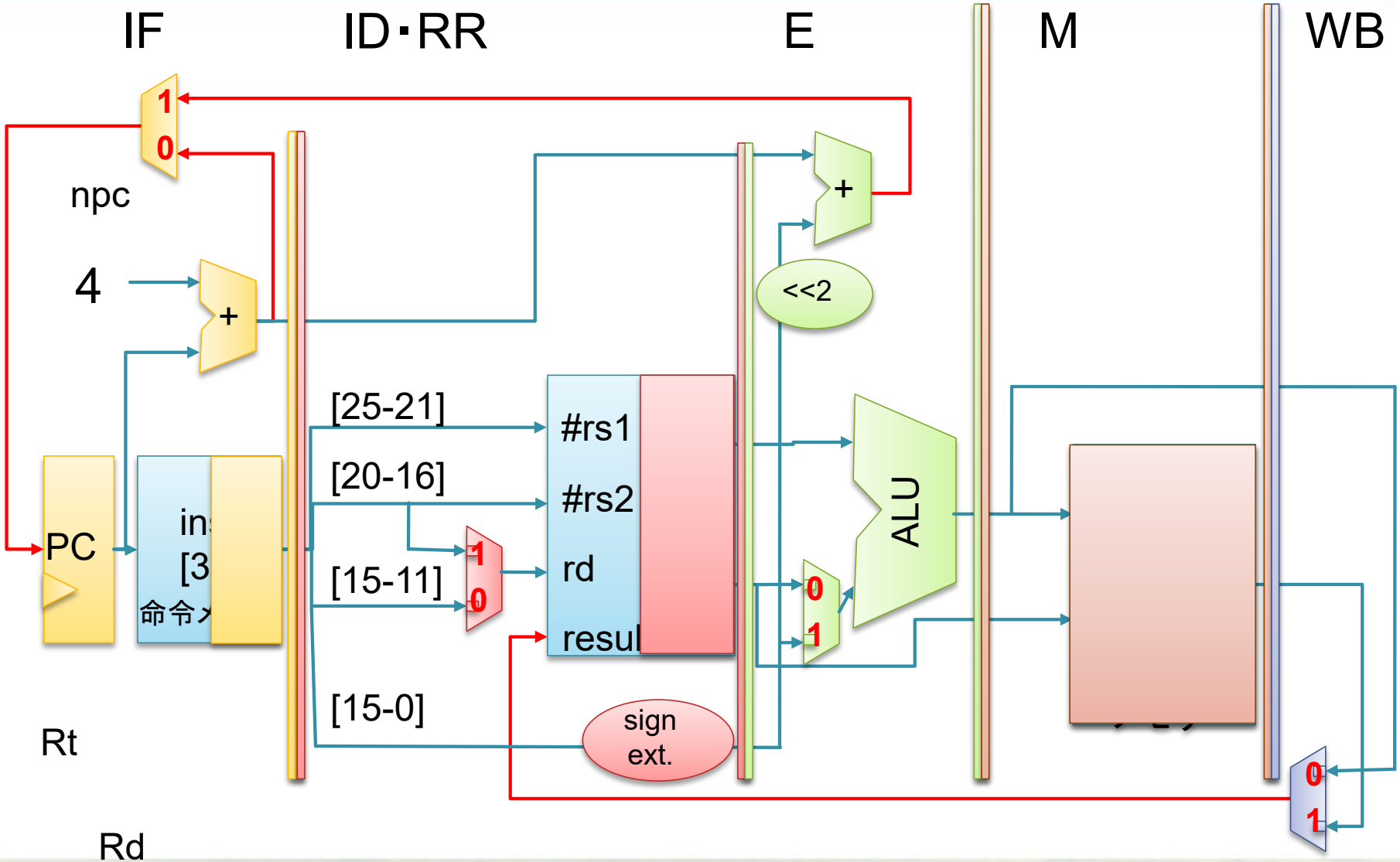
実行ステージ


IF
ID・RR
E
M
WB
1
0
npc
4
+
PC
in
[3
命令メ
[25-21]
[20-16]
[15-11]
[15-0]
Rt
Rd
#rs1
#rs2
rd
resul
1
0
sign
ext.
+
<<2
0
1
ALU
データ
メモリ
0
1

メモリステージ


IF
ID・RR
E
M
WB
1
0
npc
4
+
PC
in
[3
命令メ
[25-21]
[20-16]
[15-11]
[15-0]
#rs1
#rs2
rd
resul
1
0
sign
ext.
+
<<2
0
1
ALU
0
1
Rt
Rd

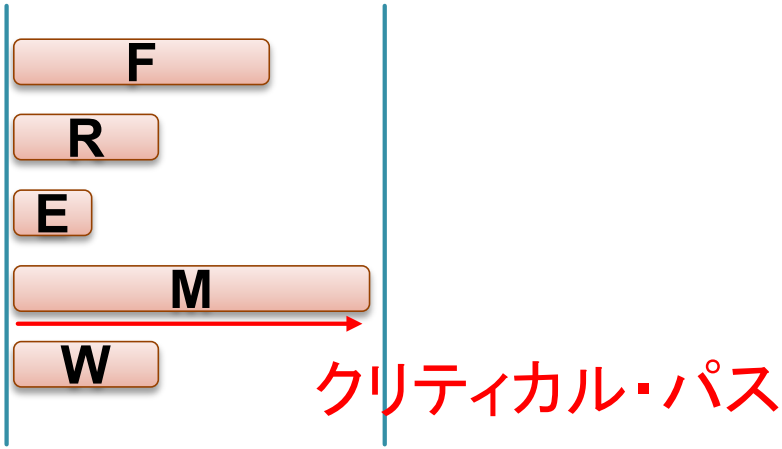
ライトバックステージ


IF
ID・RR
E
M
WB
1
0
npc
4
+
PC
in
[3
命令メ
[25-21]
[20-16]
[15-11]
[15-0]
#rs1
#rs2
rd
resul
1
0
sign
ext.
+
<<2
0
1
ALU
0
1
Rt
Rd

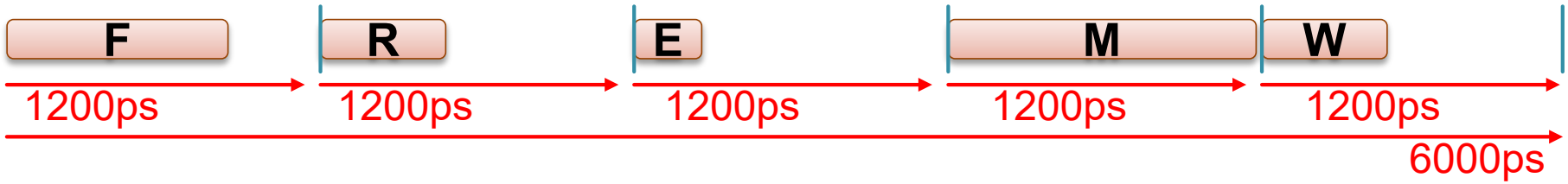
スループットとレイテンシ

- 例えば各ステージの遅延が以下の場合

フェッチ： 800ps
レジスタリード: 400ps
エクセキュート: 200ps
メモリ : 1200ps
レジスタライト: 400ps



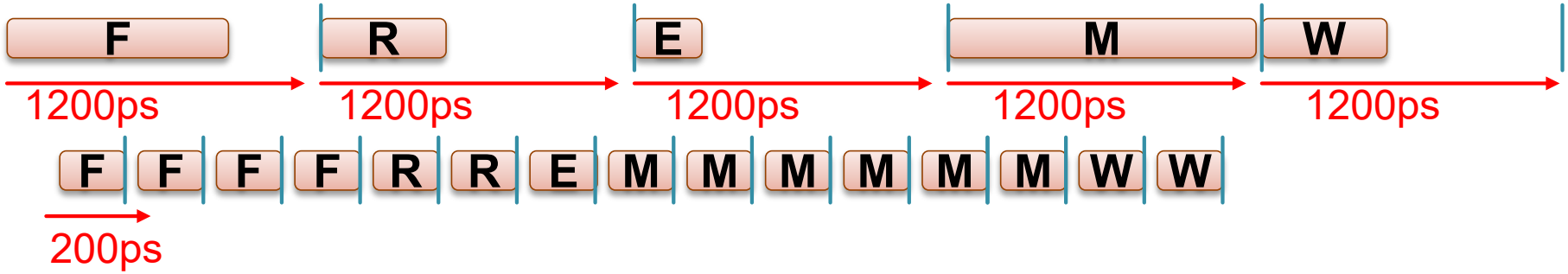
動作クロックはクリティカルパスが決定



オーバヘッドなど考慮しなければ
レイテンシ2倍
スループット2.5倍 : 性能向上

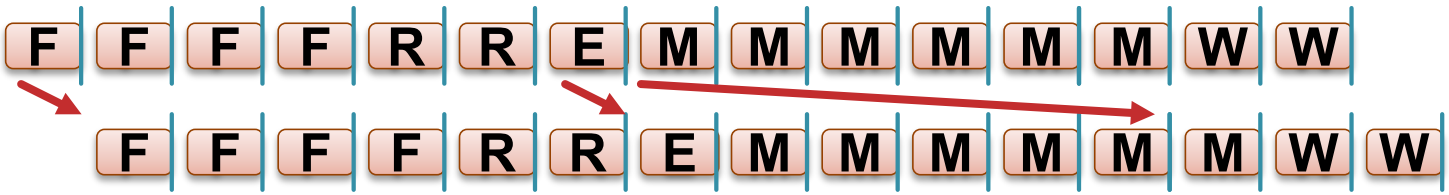
スーパ・パイプラインとクリティカルループ

- 先ほどの例は，各ステージを分割すれば…



スループット さらに6倍

- 「クリティカル・ループ」
  - 1サイクルでできないと命令間で依存があるときにバブルの影響が大きい処理
    - ≒クリティカル・パスを定める部分
  - スカラの場合の典型的なポイント



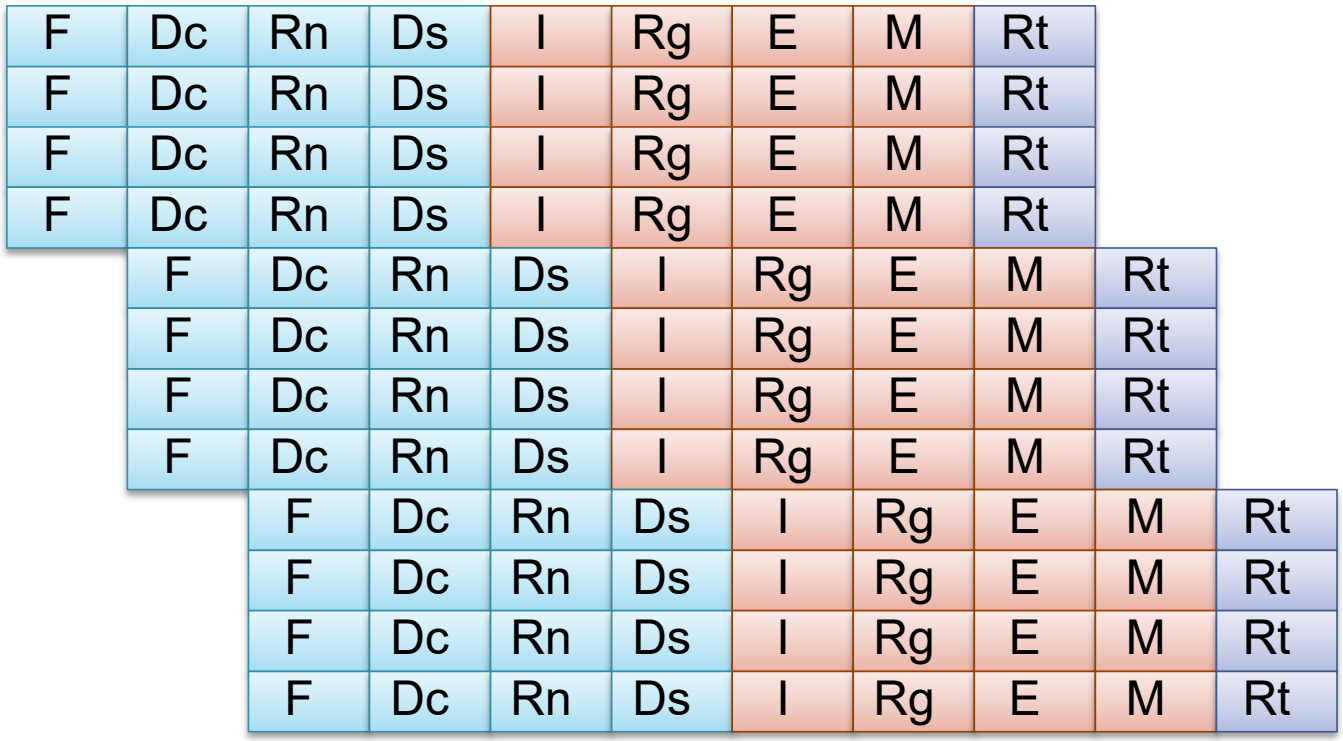
スーパ・パイプライン拡張のオーバヘッド

- **リソース効率**を落としてくる要素
  - **電力**　fV^2に比例，＋パイプラインレジスタ増加分
  - **クロッキングオーバヘッド**
    - スキュー，ジッター，ラッチオーバヘッド
  - **バブル**
    - スカラの場合特にメモリ

- 1stageの限界は**6-8FO4** [Hishikesh+, 2002]
  - Prescottの**31段**をピークに**20段**ほどまで減少

スーパ・スカラ

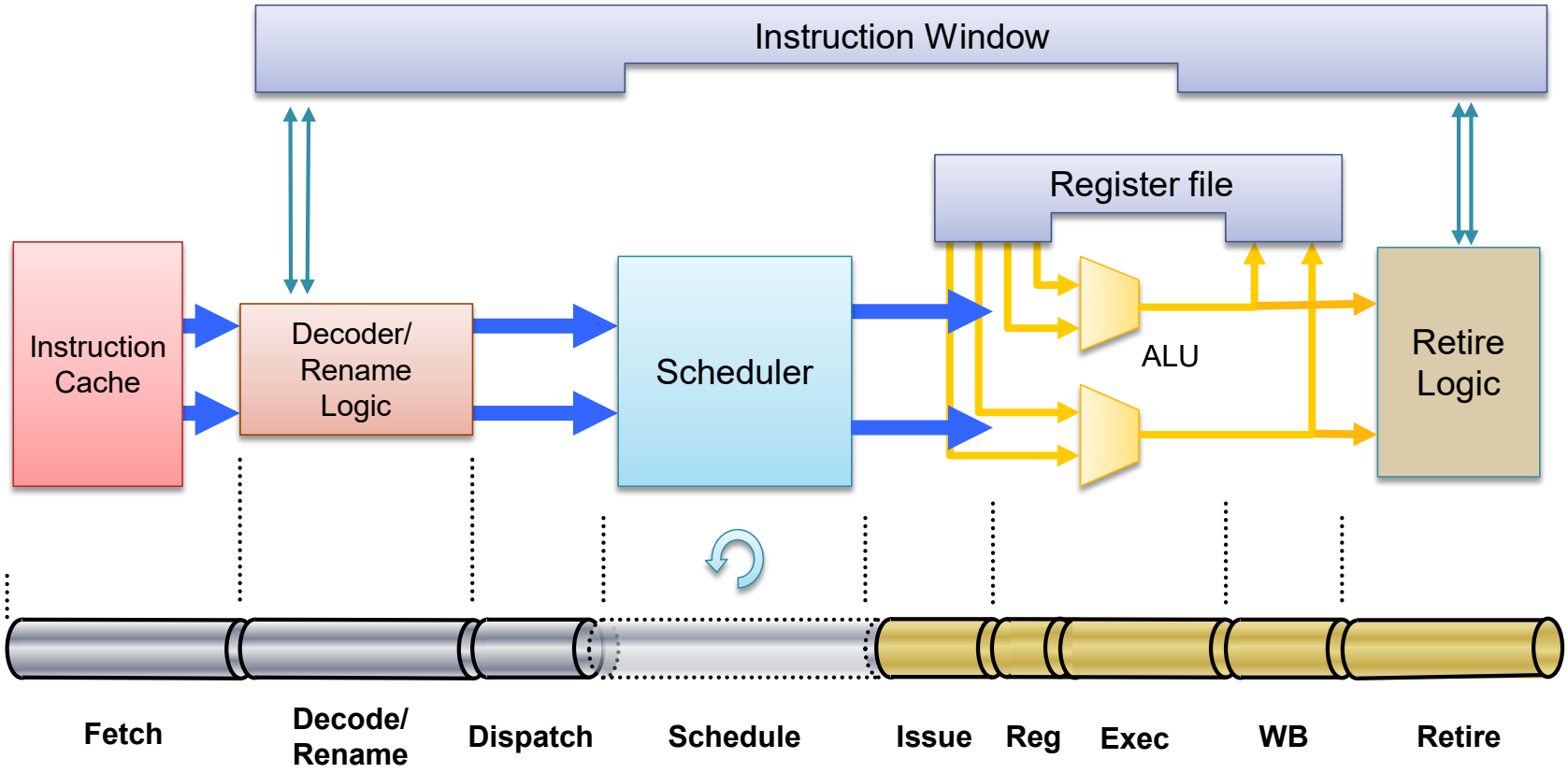
- 1サイクルに複数命令を同時処理
  - 性能ポテンシャルは　深さ×**幅**　で決定
    - **Cycle Per Instruction** から**IPC**へ



イン・オーダとアウト・オブ・オーダ

- **イン・オーダ**実行
  - コードの順番通りに実行
- **アウト・オブ・オーダ**実行(**OOO**)
  - 実行結果が不変の範囲で**処理の順番を変更**
    - キャッシュミス時，依存のない後続命令を発行可能
    - 浮動小数点演算を効率良くオーバラップ
    - vs. ソフトウェアスケジューリング
      分岐を越えたスケジューリングが可能
  - ハードウェアは複雑化するが性能向上

アウト・オブ・オーダ・スーパスカラ・プロセッサ

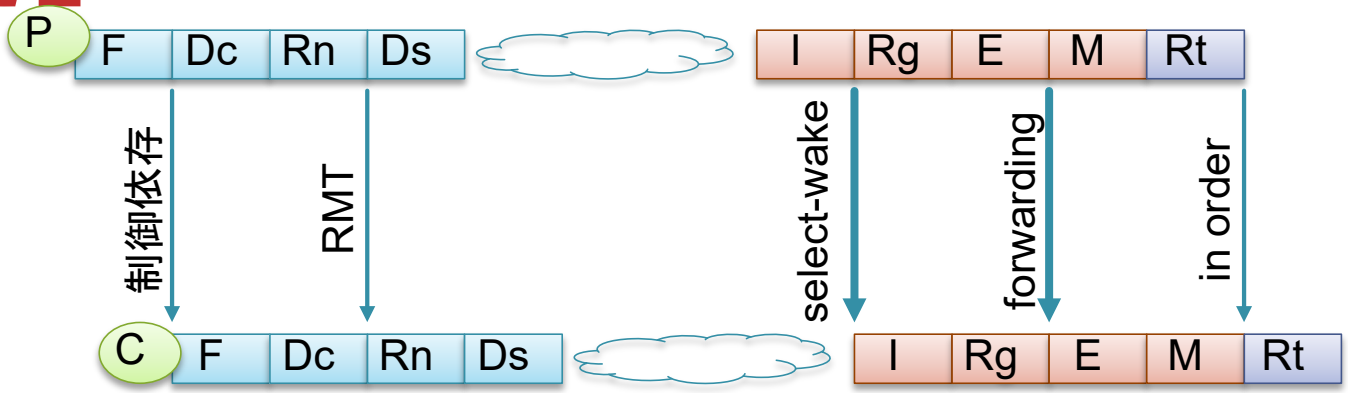


スーパスカラ・プロセッサの利点

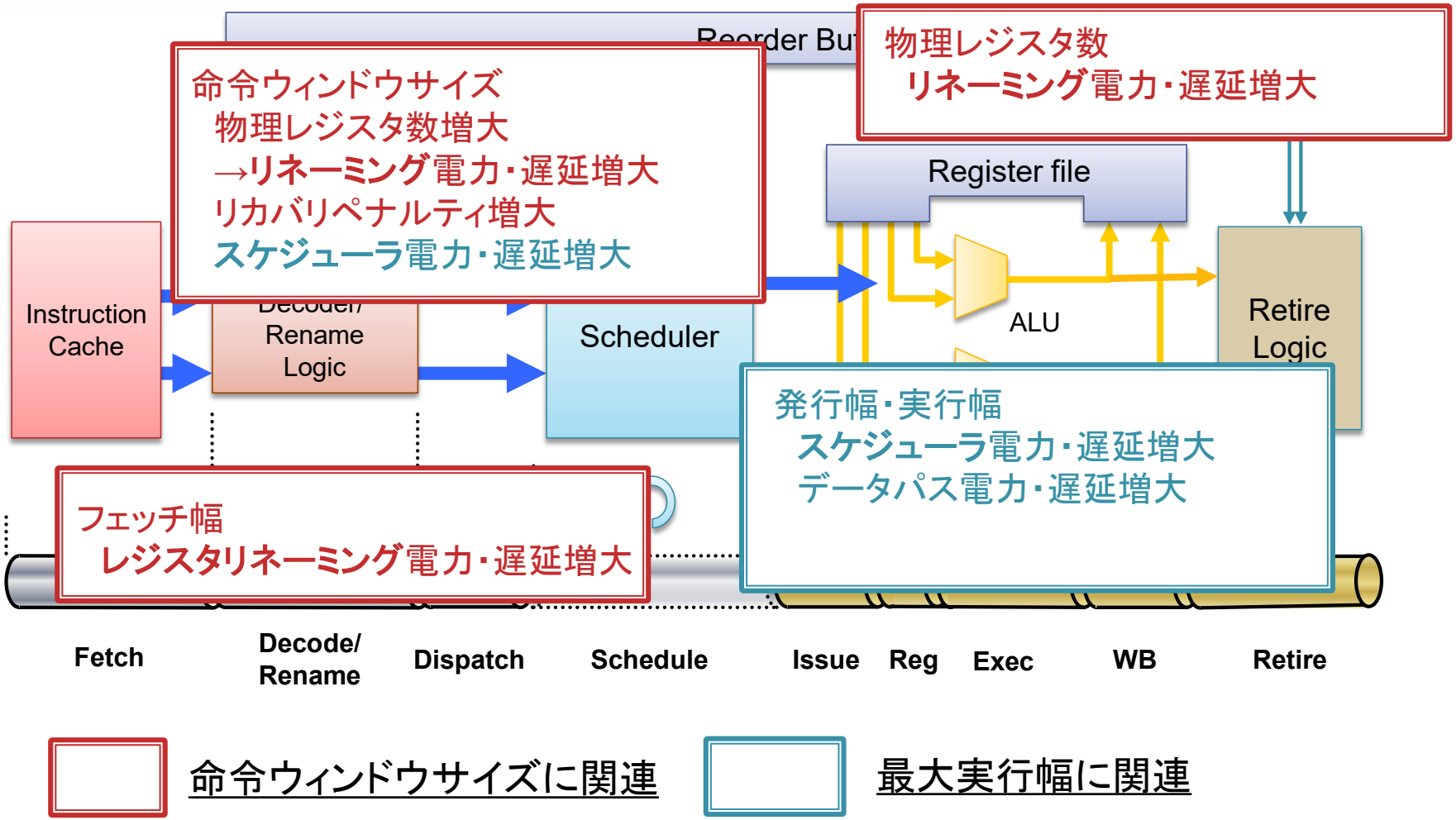
- **データ依存**や**制御依存**が**単純でない**プログラムを最も高速(効率的)に実行できるアーキテクチャ
  - 命令単位の制御依存，データ依存に対応
  - データフロー制約をハードウェアが解析して
    ハードウェア毎に適したILP実行
- 現状ほぼ**唯一のオプション**

スーパスカラのクリティカル・ループ

- 1サイクルで終わらせないといけない処理
- 問題となるのは
  - **制御依存**
  - **レジスタ・リネーミング**
  - **発行ループ**
  - **データフォワーディング**
  - **リタイア処理**



OoOスーパスカラアーキテクチャの隘路



STRAIGHTプロセッサ

2018/11/12

Motivation

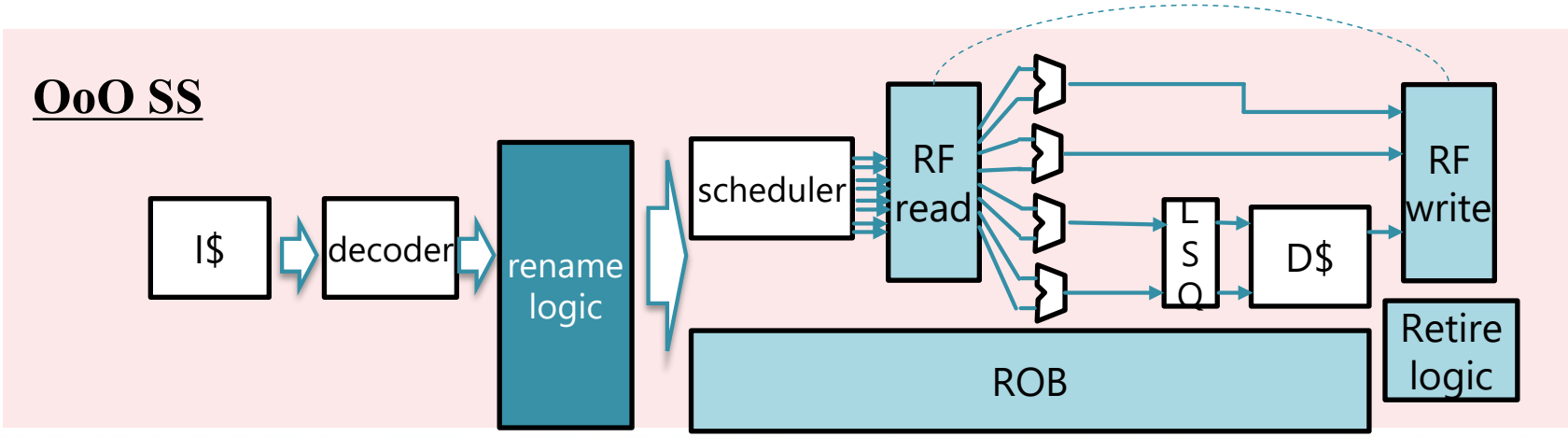
Single-thread Execution Mechanism

the most powerful
the most common

⇩

for the single thread performance

- Out-of-order superscalar (**OoO SS**) architecture
  - At the heart of the architecture is **Register renaming**
    - Resolves false-dependency
    - Enables to hold speculative values for recovery



The Reported Problems of Register Renaming

- **Its complex hardware mechanism raises the overhead**

1. **Power and area budgets ☹**

   ***"4th-highest power density on the chip"* (P6)**

   ***"Larger than L1D cache"* (R. Preston+, 2002)**

2. **Frontend scalability ☹**

   ***Delay increases in proportion to the width²* (S. Palacharla+, 1997)**

3. **Miss prediction recovery penalty ☹**
   (Reorder buffer (ROB) scalability☹)

   ***"17.2cycles in average"* (S. Petit+, 2014)**

The Mechanism of Register Renaming

- A mechanism for **resolving false-dependences**

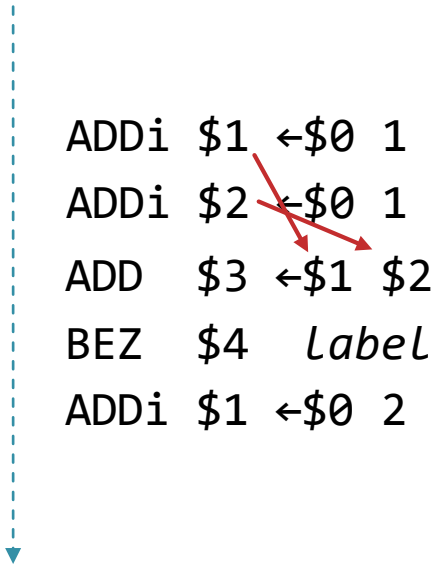
instruction sequence

```
ADDi $1 ←$0 1
ADDi $2 ←$0 1
ADD  $3 ←$1 $2
BEZ  $4  label
ADDi $1 ←$0 2
```

The Mechanism of Register Renaming

- A mechanism for **resolving false-dependences**

instruction sequence



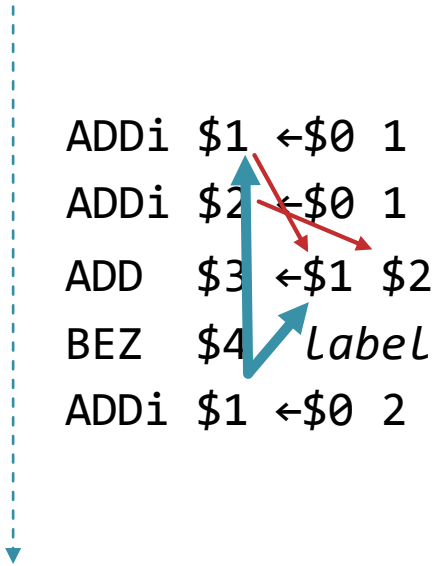
```
ADDi $1 ←$0 1
ADDi $2 ←$0 1
ADD  $3 ←$1 $2
BEZ  $4  label
ADDi $1 ←$0 2
```

true dependencies → resolved by the scheduler

The Mechanism of Register Renaming

- A mechanism for **resolving false-dependences**

instruction sequence



true dependencies → resolved by the scheduler

false dependencies
→ essentially, can be executed in OoO

The Mechanism of Register Renaming

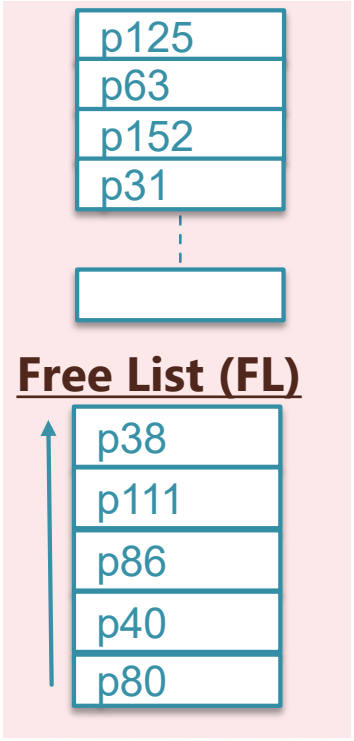
- A mechanism for **resolving false-dependences**

instruction sequence

```
ADDi $1 ←$0 1
ADDi $2 ←$0 1
ADD  $3 ←$1 $2
BEZ  $4  label
ADDi $1 ←$0 2
```

Typical register renaming mechanism (RAM based)

Register Map Table (RMT) or Register Alias Table(RAT)

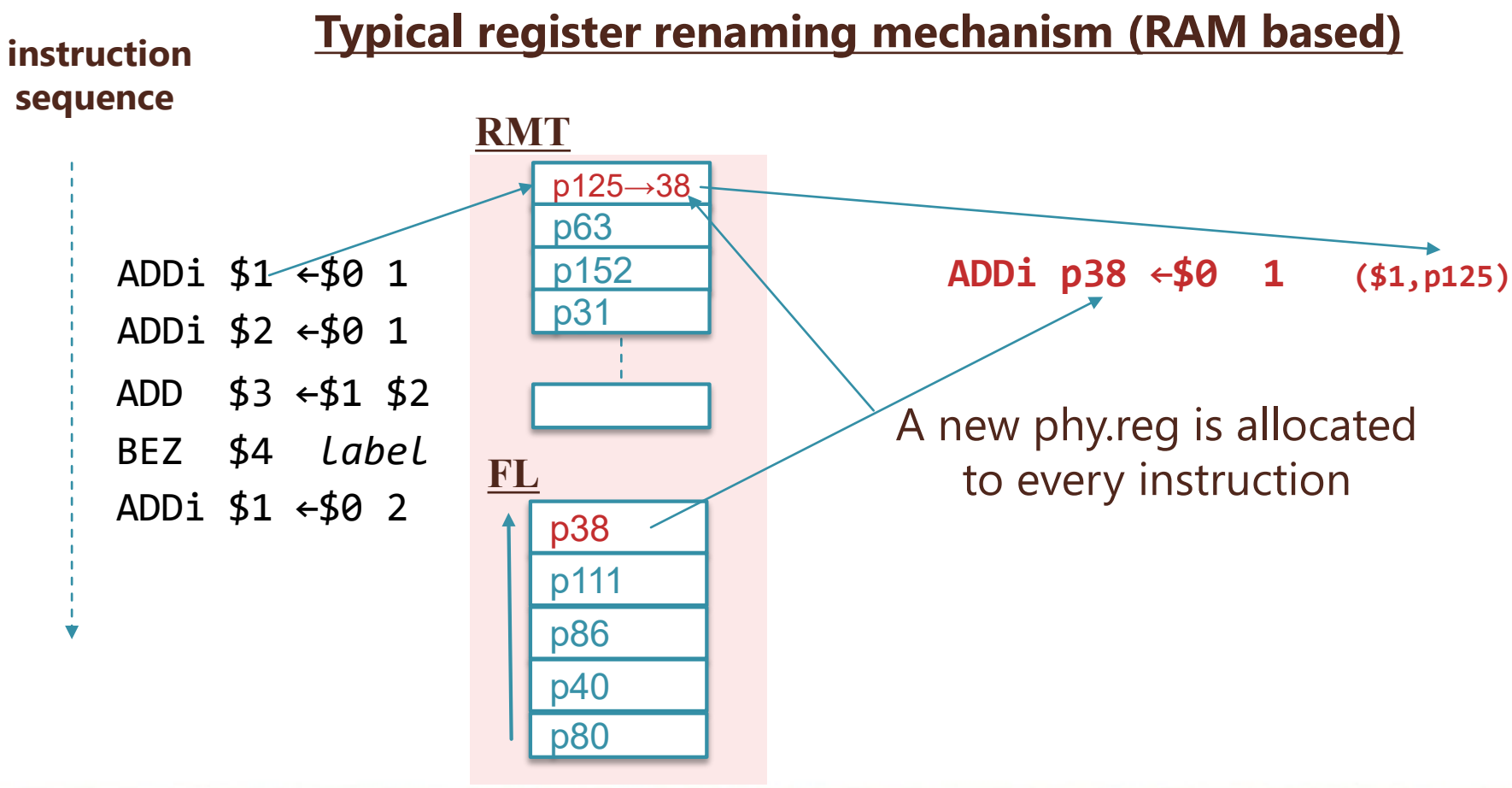


Maps logical registers to the internal physical registers (**phy.reg.**)

Holds the free register numbers in a queue structure

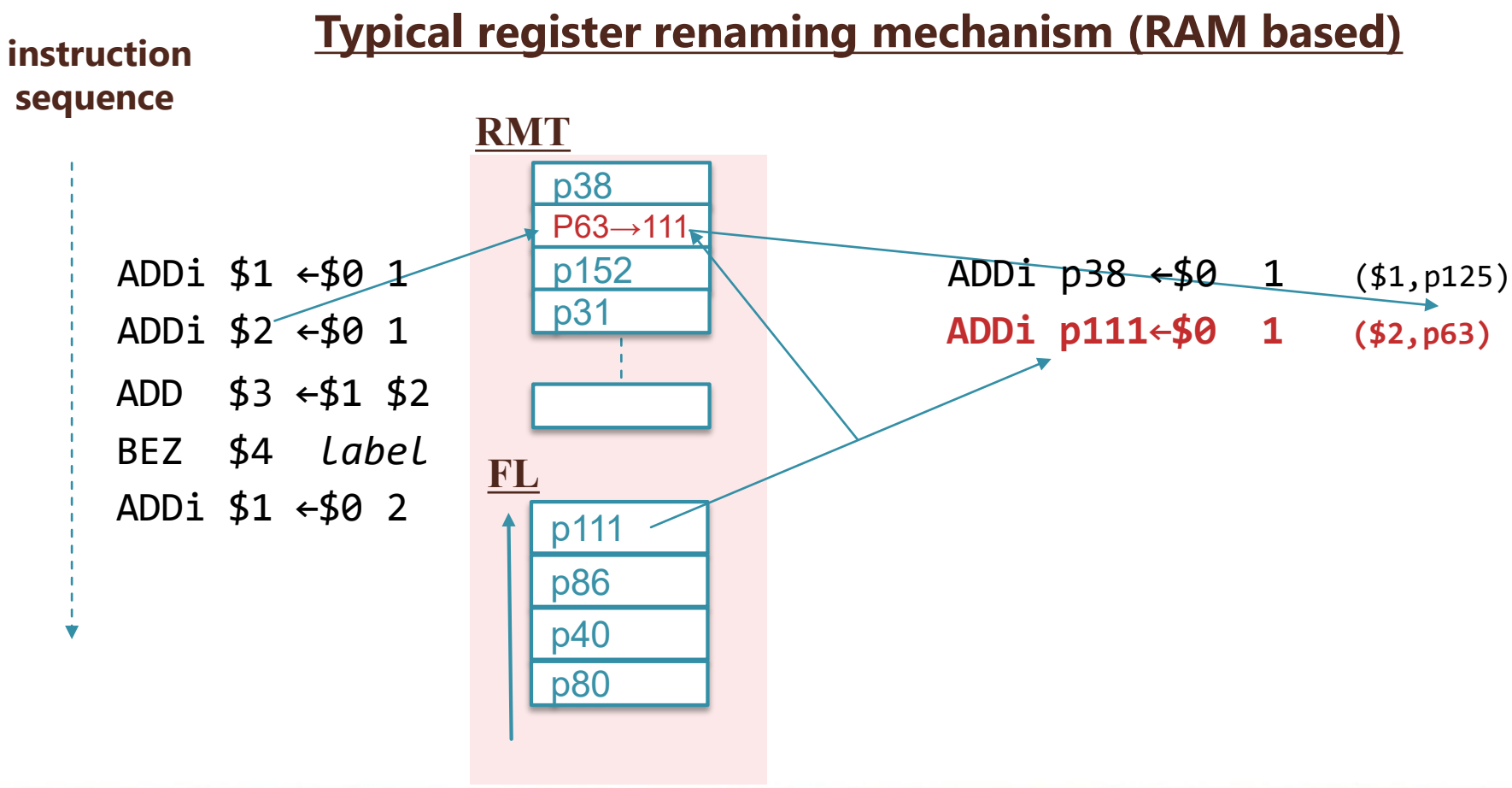
The Mechanism of Register Renaming

- A mechanism for **resolving false-dependences**



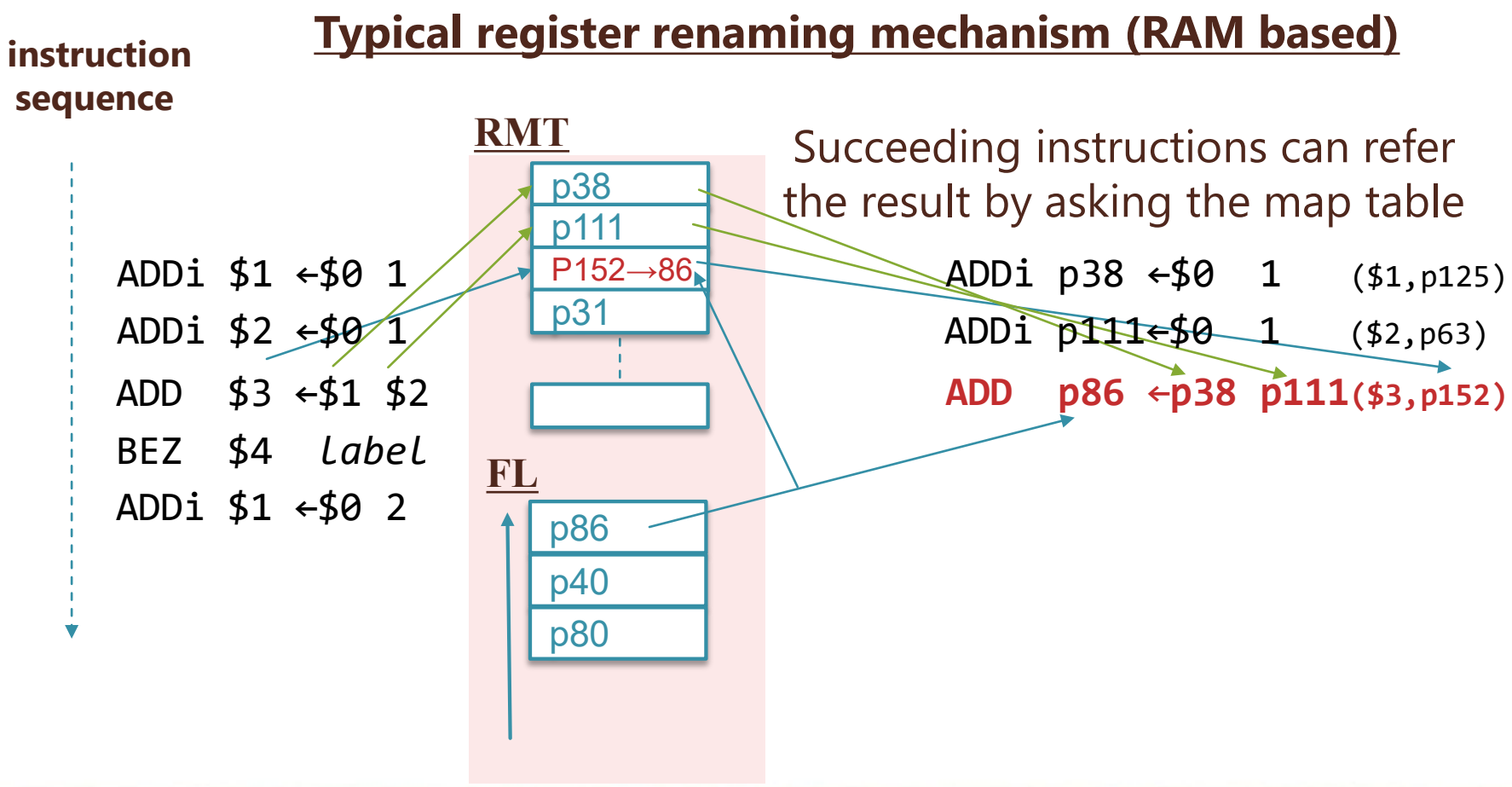
The Mechanism of Register Renaming

- A mechanism for **resolving false-dependences**



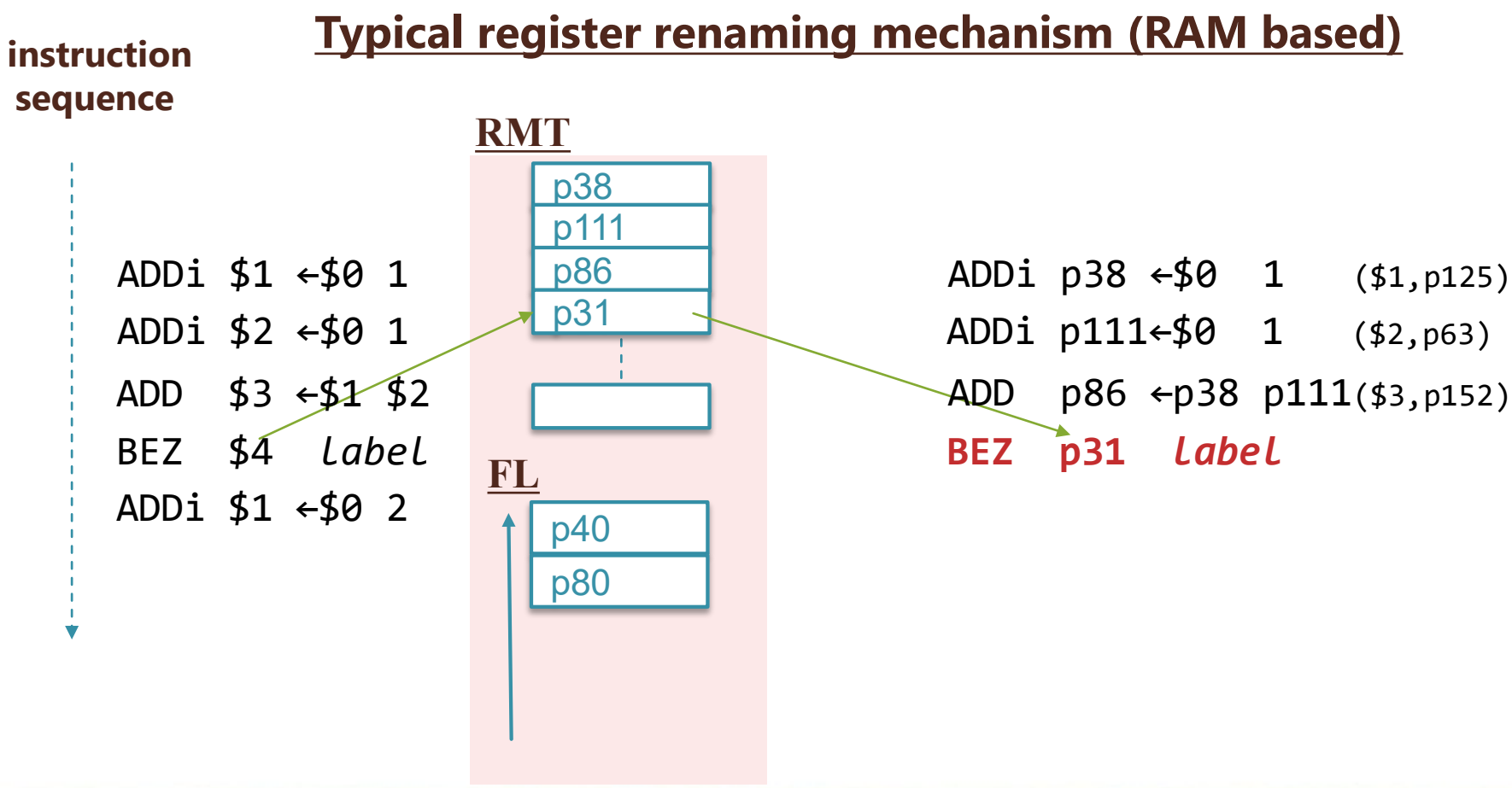
The Mechanism of Register Renaming

- A mechanism for **resolving false-dependences**



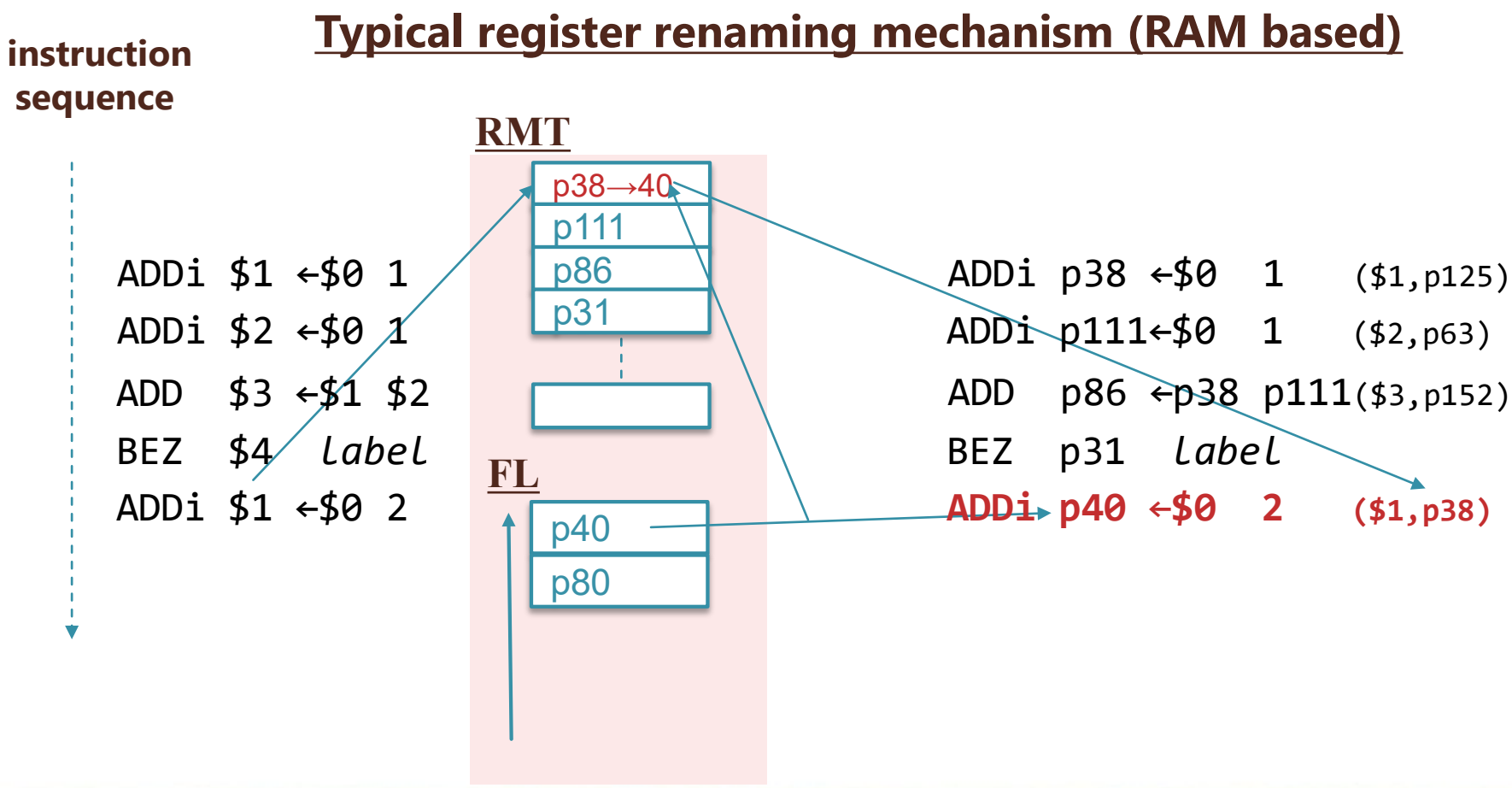
The Mechanism of Register Renaming

- A mechanism for **resolving false-dependences**



The Mechanism of Register Renaming

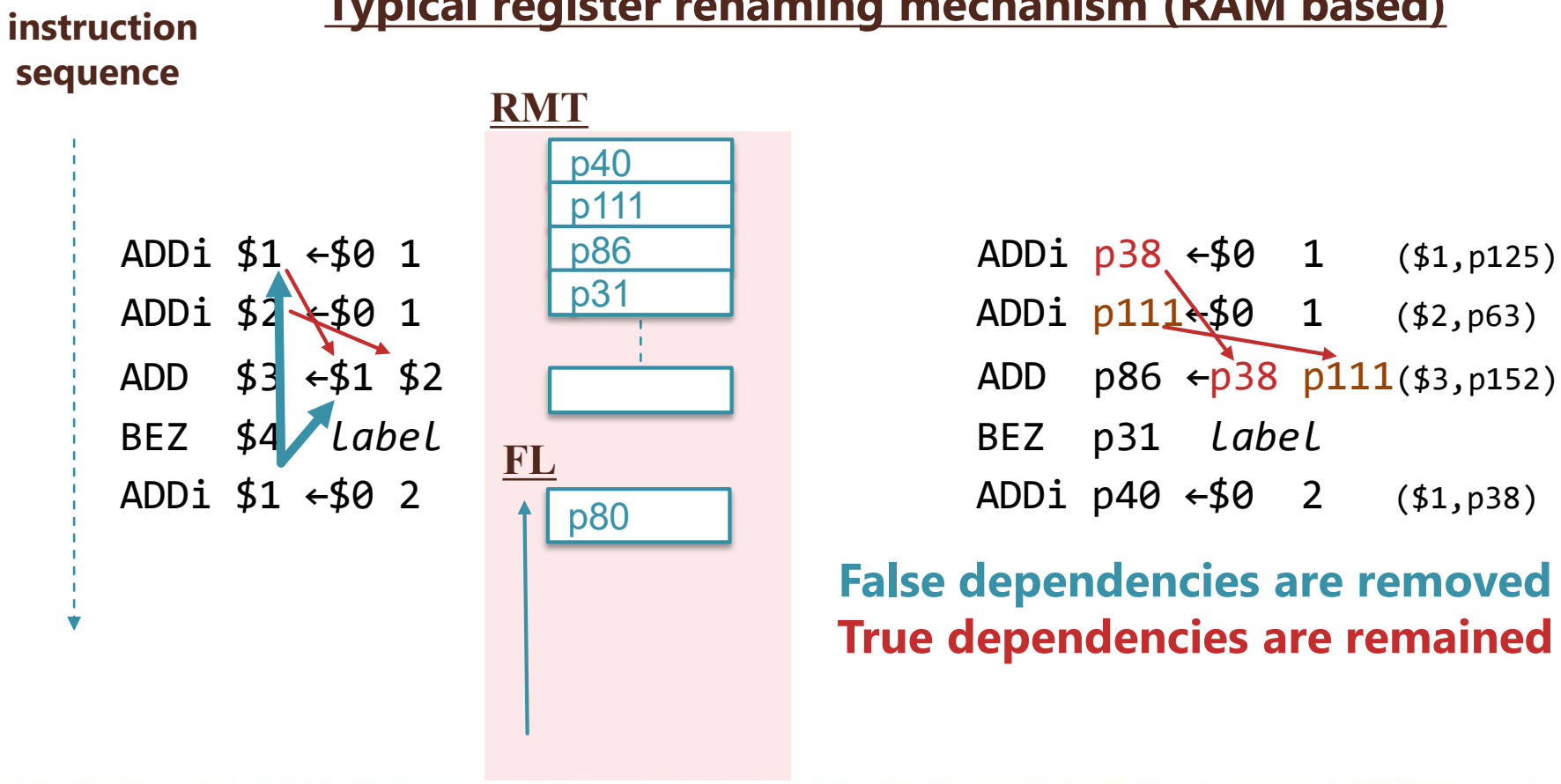
- A mechanism for **resolving false-dependences**



The Mechanism of Register Renaming

- A mechanism for **resolving false-dependences**

Typical register renaming mechanism (RAM based)

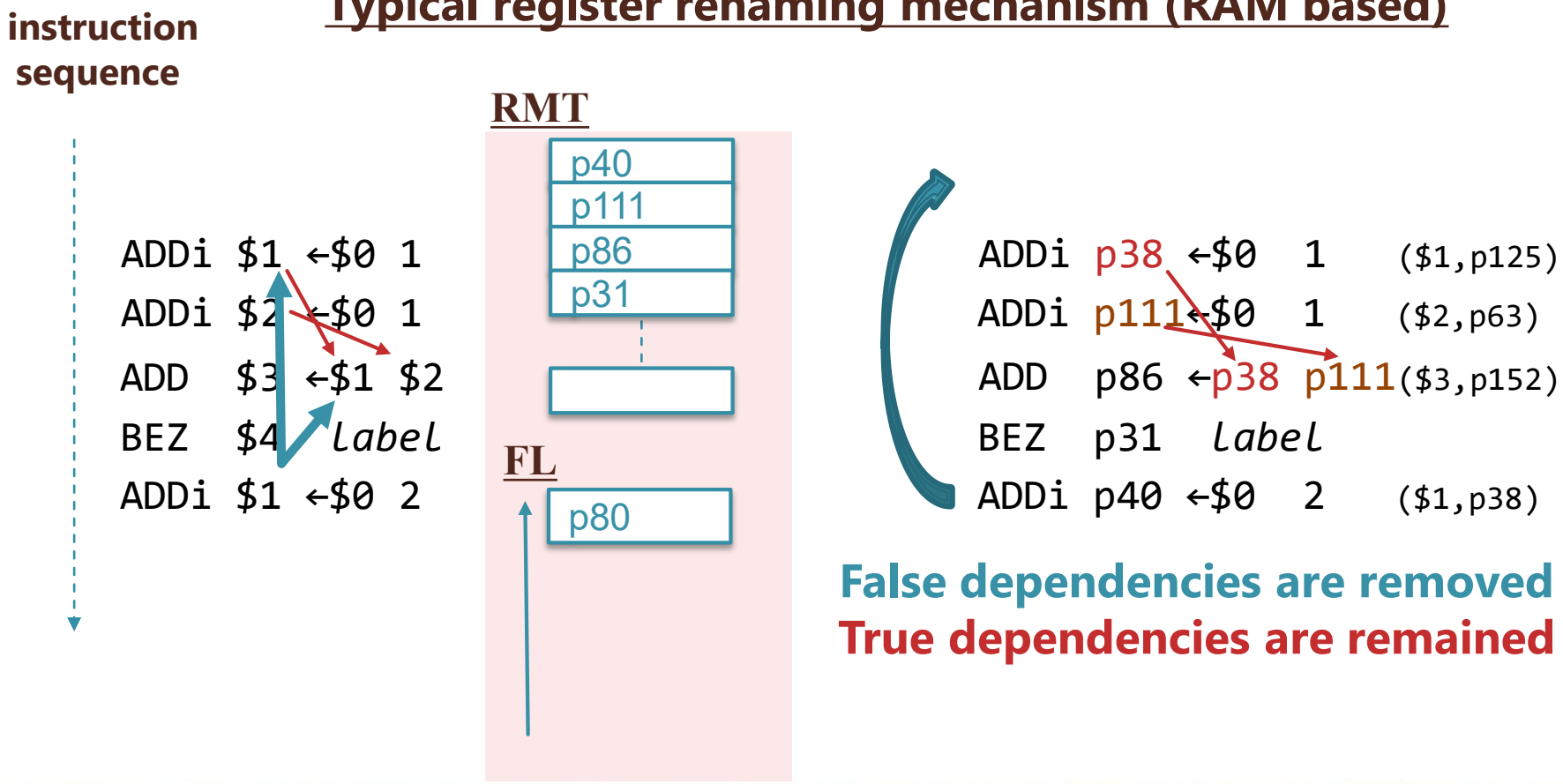


False dependencies are removed
True dependencies are remained

The Mechanism of Register Renaming

- A mechanism for **resolving false-dependences**

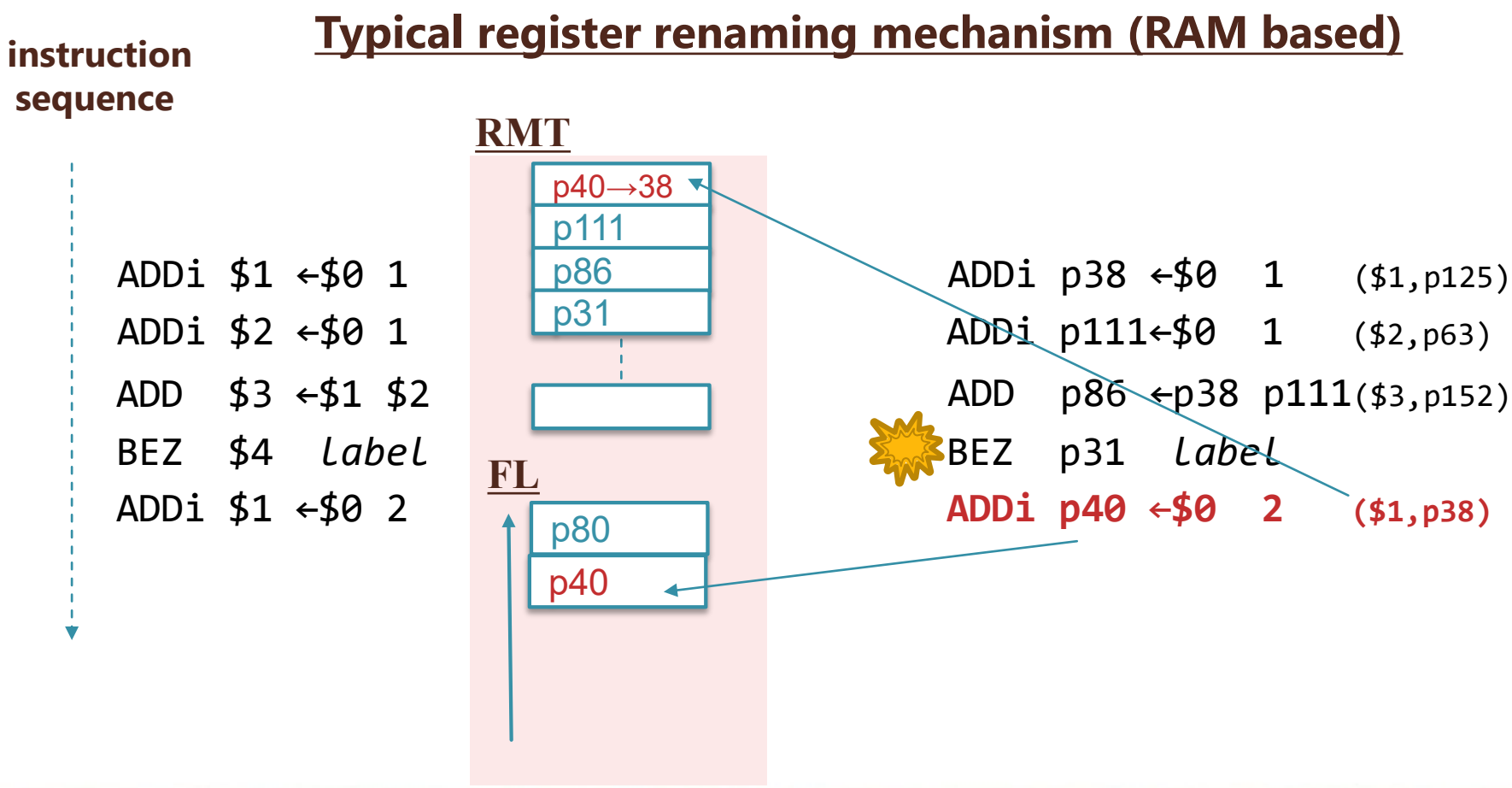
Typical register renaming mechanism (RAM based)



False dependencies are removed
True dependencies are remained

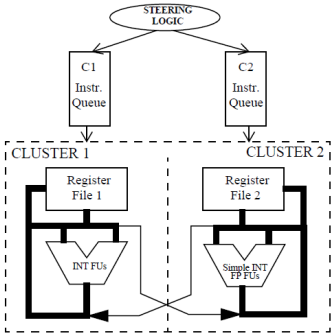
The Mechanism of Register Renaming

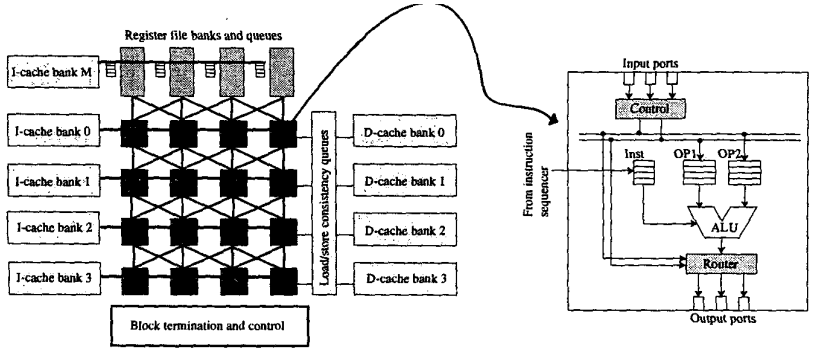
- The mechanism also enables the **miss recovery**



かつてのILPアーキテクチャの関連研究[90s後半-00s]

- **周波数向上が限界**なので**最大実行幅拡張**を模索
  - クラスタ化/ タイル/ 投機マルチスレディング/ VLIW・・・

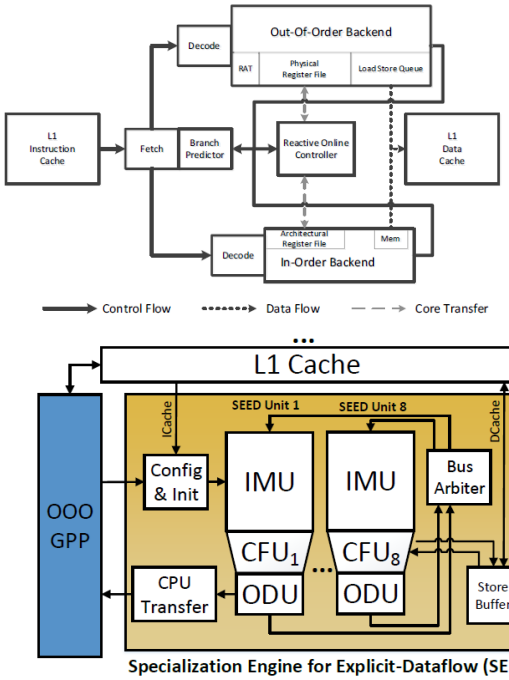




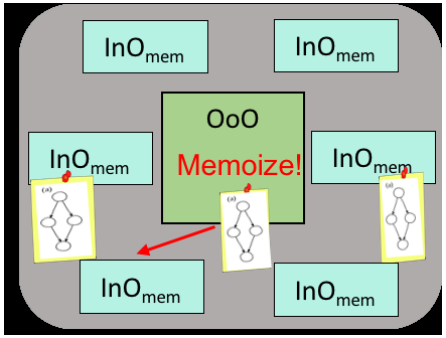
- **スループット用途**であれば**他アプローチが有利**
  - **複雑化したアーキテクチャ**にリソースを投入してもマルチコアほどコスト効果は目立たない
  - マルチコアやアクセラレータにソフトウェアが適応

最近のILPアーキテクチャの関連研究

- **Composite Core**
  [Lukefahr+, 2012]
  - InOとOoO方式を切り替えて高効率
- **SEED** [Nowatzki+, 2015]
  - OoO方式とタイルアーキテクチャを切り替えて高効率
- **Mirage Core** [Padmanabha+, 2017]
  - 少数のOoOコアでスケジュール，多数のInOコアで実行



- いずれもOoOスーパスカラが**最高性能**



RTC，逆デュアルフロー

- リネーミングの電力や遅さに対策
  - 結果をキャッシュしてスキップ
  - RTC[Vajapeyam+,1997]
    - ヒット率は30%程度
  - GRTC（逆デュアルフローアーキテクチャ）[Shioya+,2014]
    - 命令間の距離が（同一コントロールフロー上では）不変であることに着目，距離と論理レジスタによる再利用可能表現を提案
    - ヒット率を60~70%まで向上

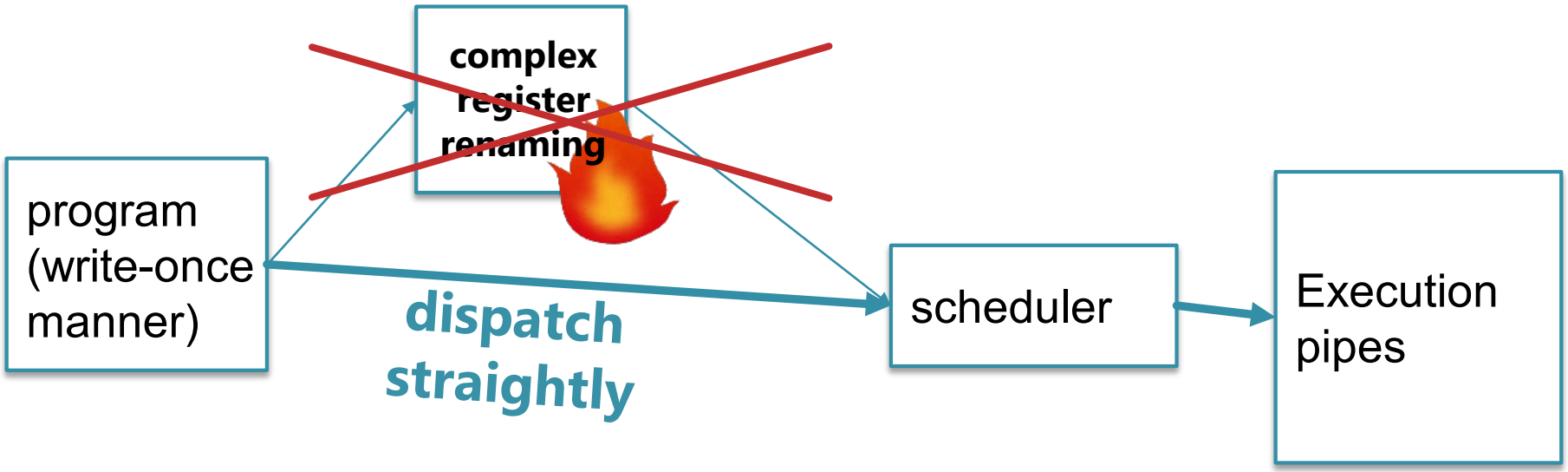
The Objective of this Work

Can we make an effective OoO architecture without register renaming?

- A possible game changer for **ILP** processor design
  - Super low power OoO execution
  - Further larger instruction window
  - Simple OoO processor that can be easily designed
- Our approach: **Instruction Set Architecture (ISA)** and **compiler** to **statically remove false-dependency**

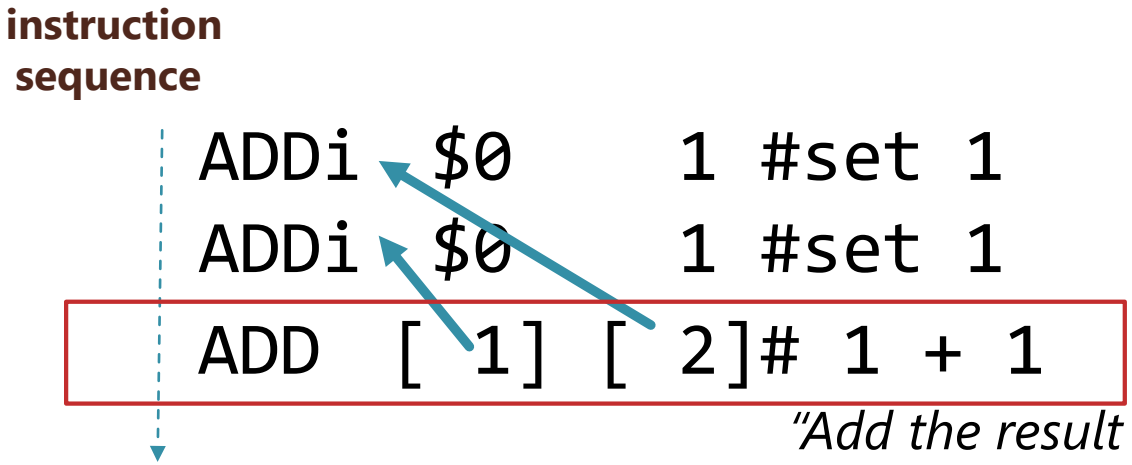
Basic Concepts

- **False-dependencies** are derived from **register overwrites**
- **If programs use registers in write-once manner…**



STRAIGHT ISA

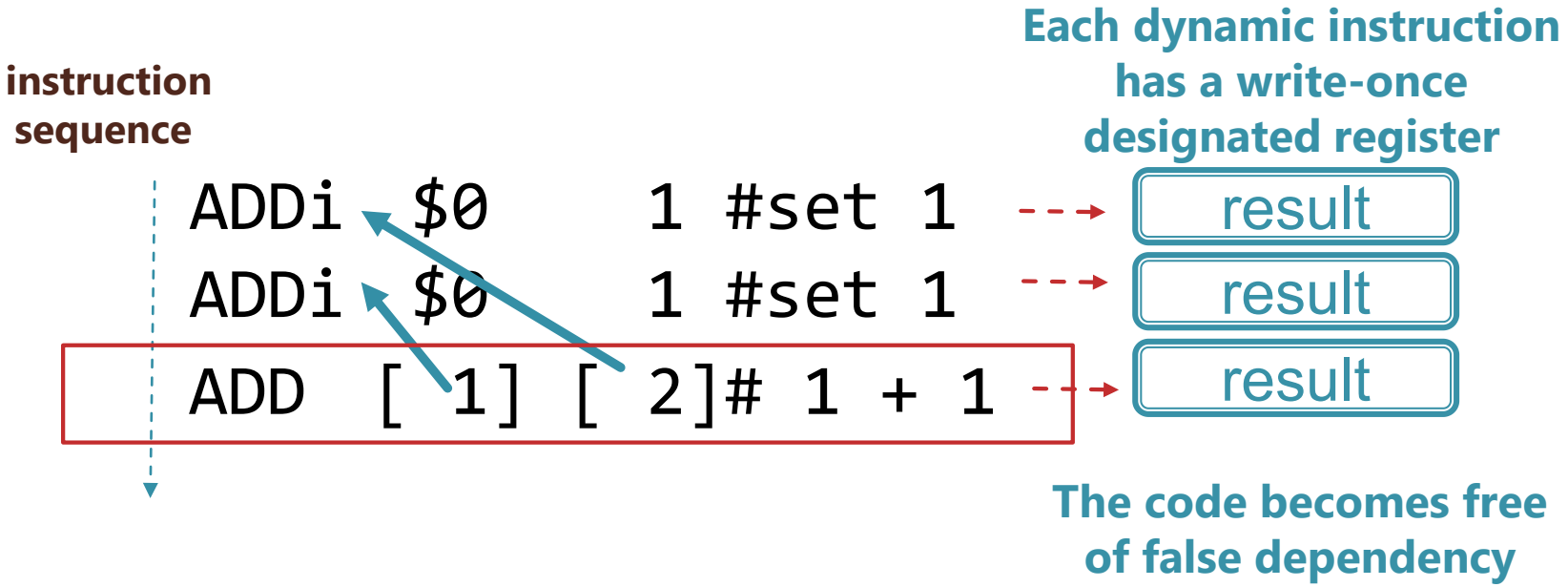
- Can ensure each register is used in write-once manner
- Expresses source operands by ***distances***
  - Instead of logical register numbers
  - Distance: dynamic instruction count from its producer



"Add the result of the previous inst. to the result of the second previous inst."

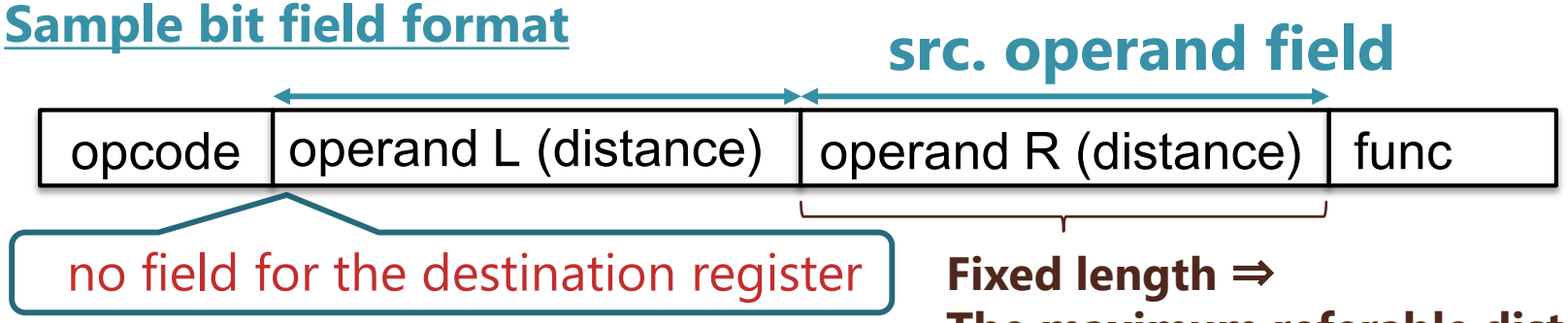
STRAIGHT ISA

- Can ensure each register is used in write-once manner
- Expresses source operands by ***distances***
  - Instead of logical register numbers
  - Distance: dynamic instruction count from its producer



Bit Field Format and the Register Life-time

Sample bit field format



Fixed length ⇒
The maximum referable distance is naturally defined
Ex. 5bit ⇒ up to 31inst.
10bit ⇒up to1023inst.

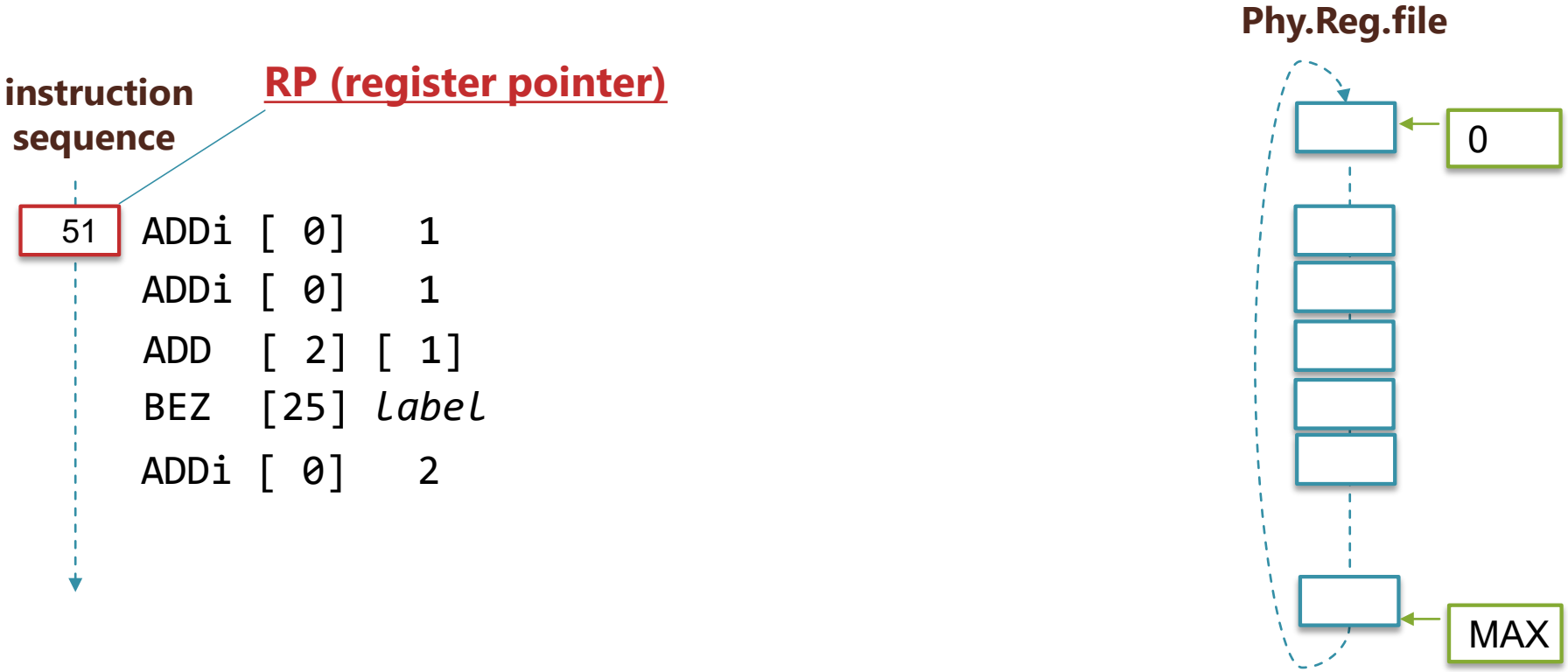
- the program is executable in practical HW
  - Values can be discarded as execution proceeds so that it won't require the infinite number of registers

Special register “Stack Pointer” (SP)

- The only rewritable register in the architecture
- **Makes the ISA practical**
  - Ensuring that any program can be theoretically written
  - (One rewritable register is sufficient to provide a safe-net)

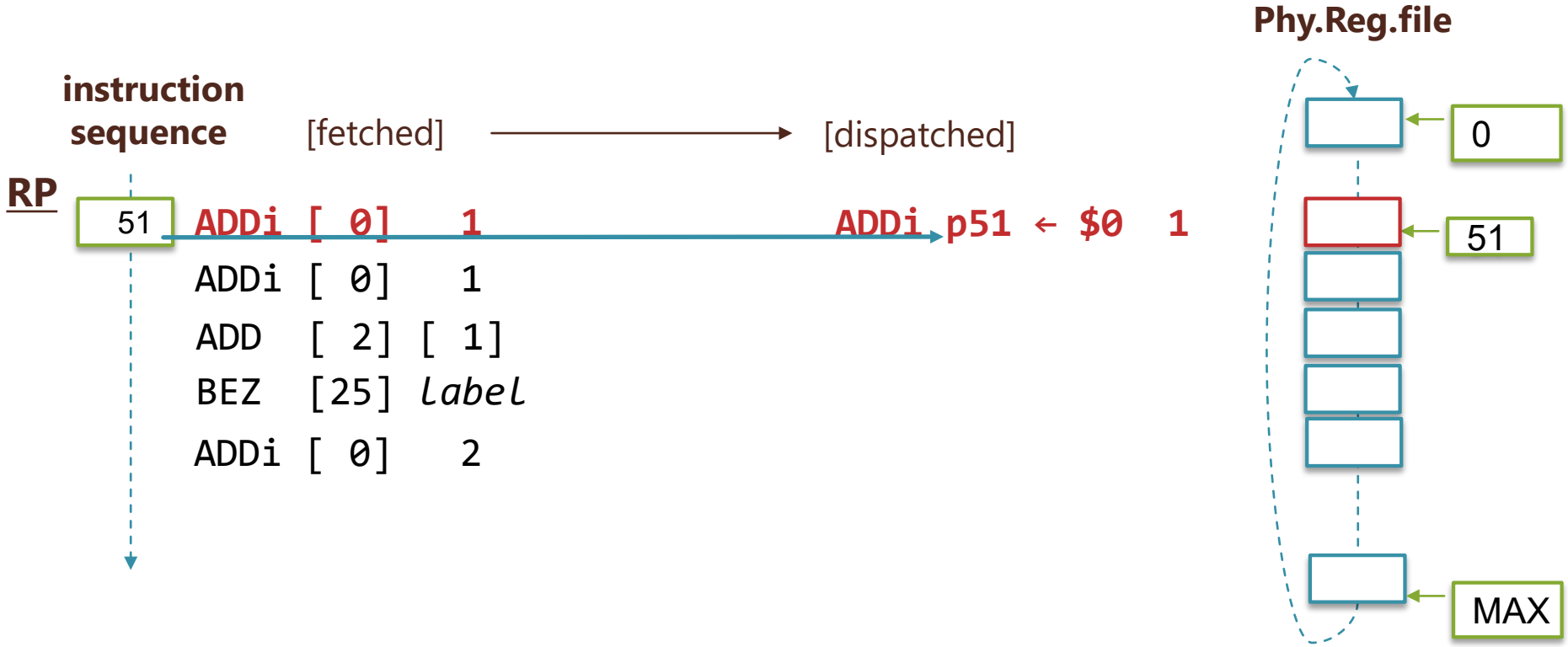
How the ISA can Simplify the Processor

- Mechanism of the operand determination
  - Conventionally processed by **register renaming**



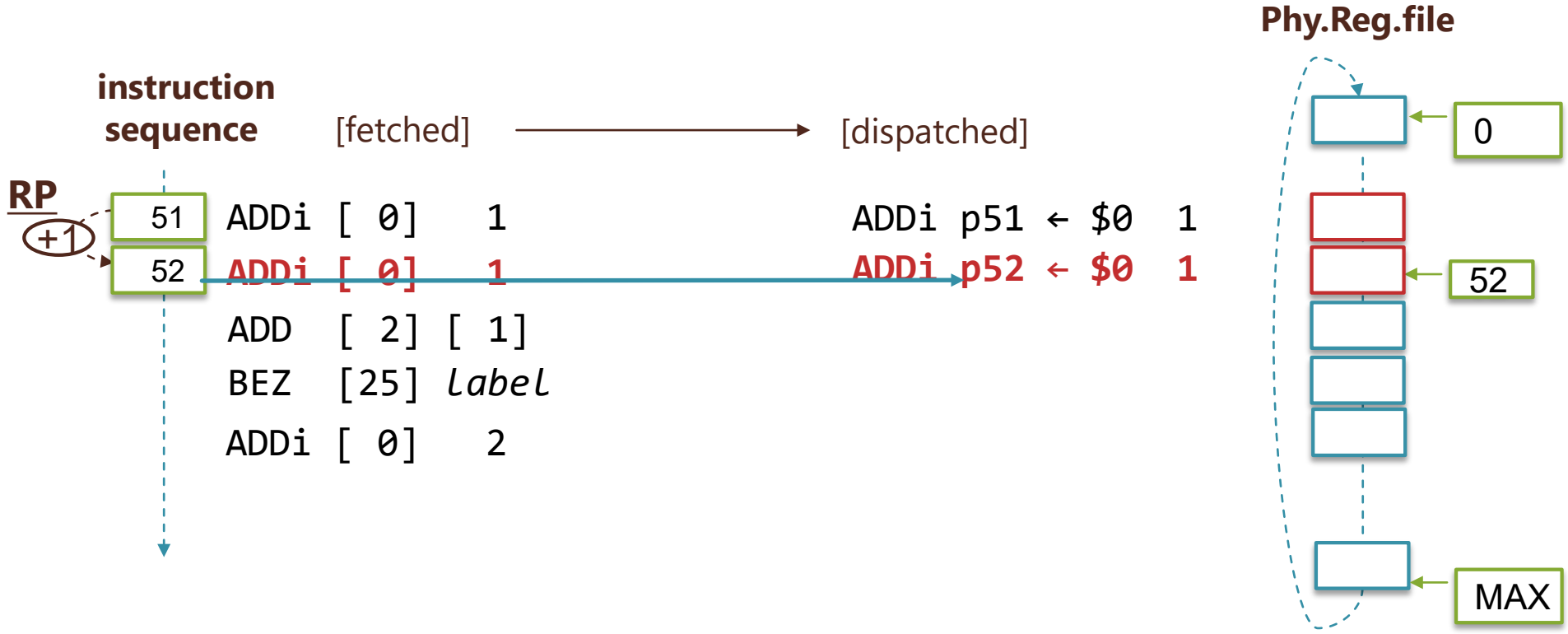
How the ISA can Simplify the Processor

- Mechanism of the operand determination
  - Conventionally processed by complex **register renaming**



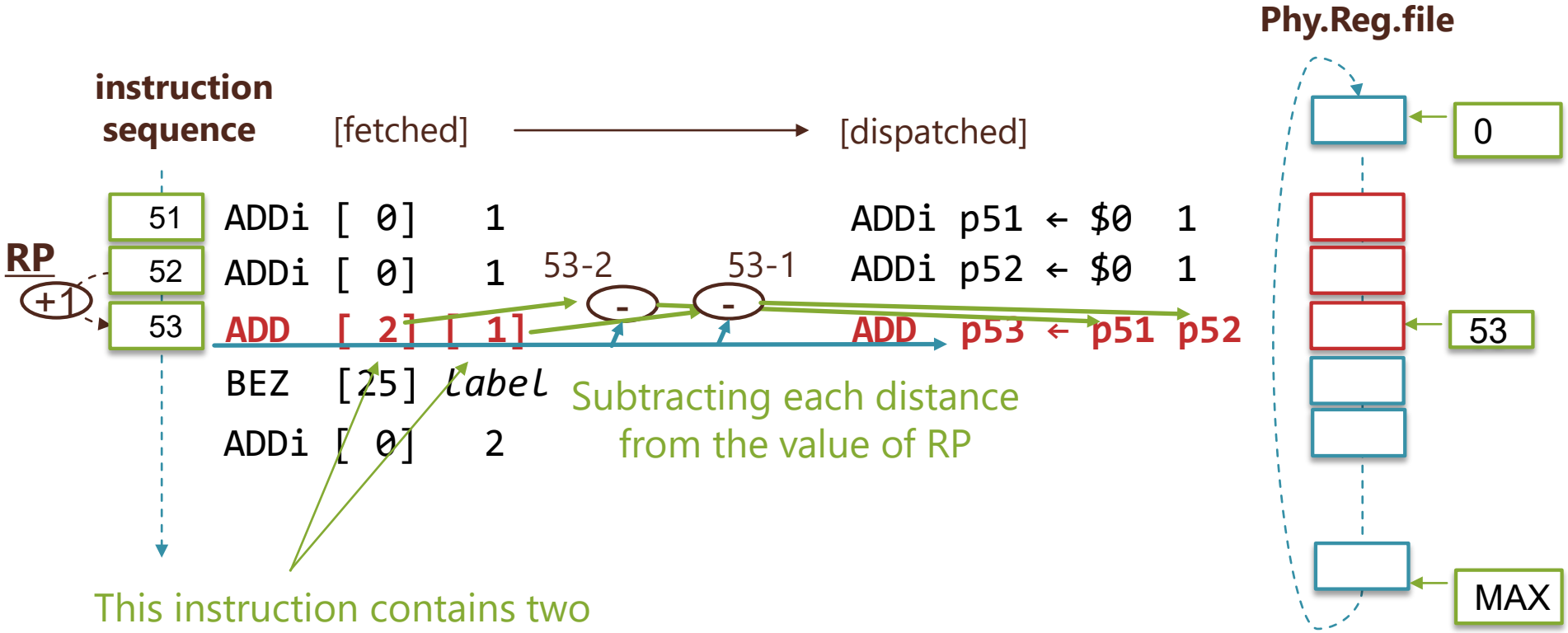
How the ISA can Simplify the Processor

- Mechanism of the operand determination
  - Conventionally processed by complex **register renaming**



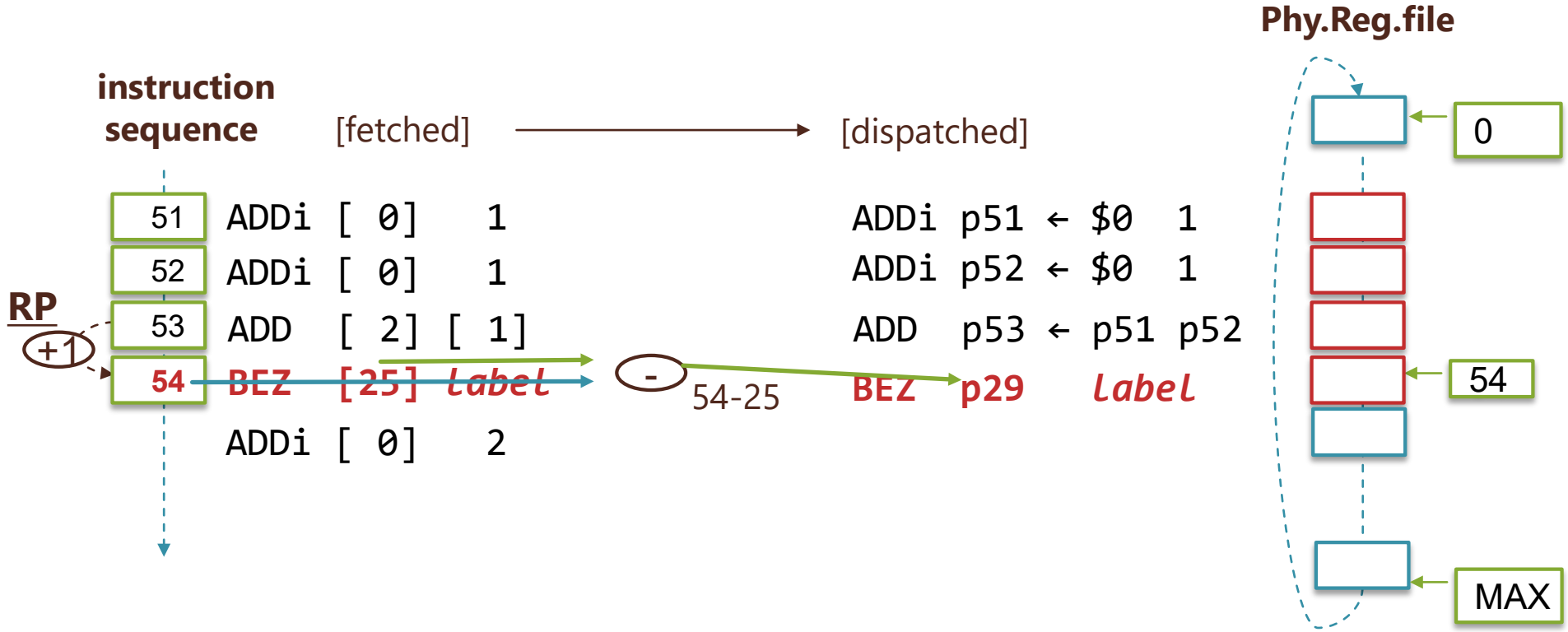
How the ISA can Simplify the Processor

- Mechanism of the operand determination
  - Conventionally processed by complex **register renaming**



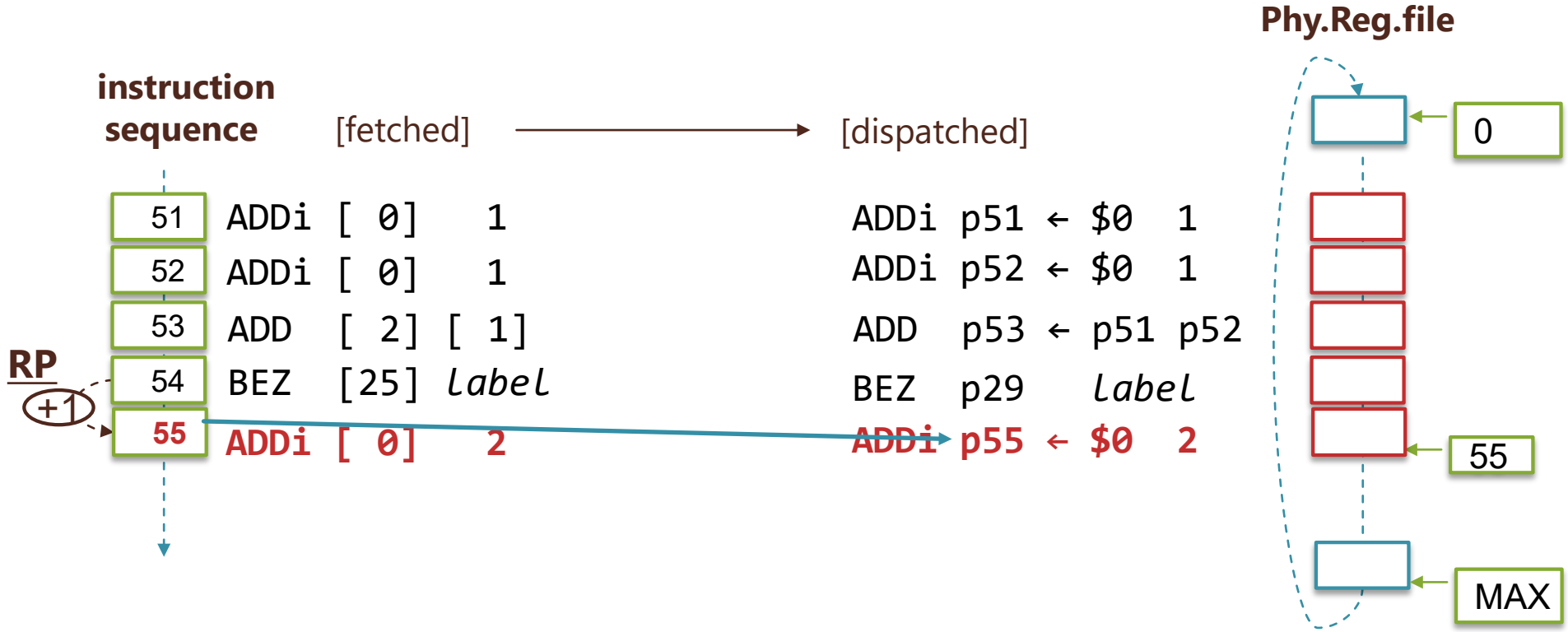
How the ISA can Simplify the Processor

- Mechanism of the operand determination
  - Conventionally processed by complex **register renaming**



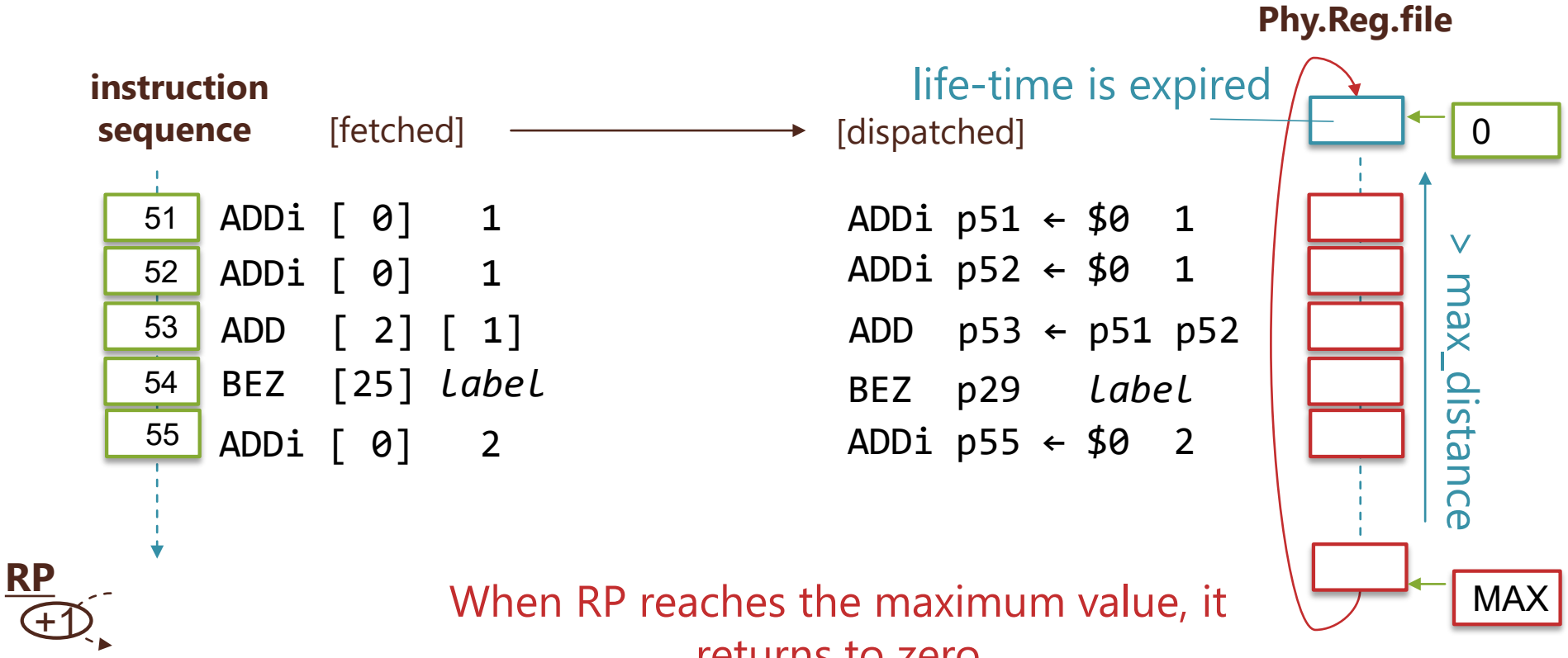
How the ISA can Simplify the Processor

- Mechanism of the operand determination
  - Conventionally processed by complex **register renaming**



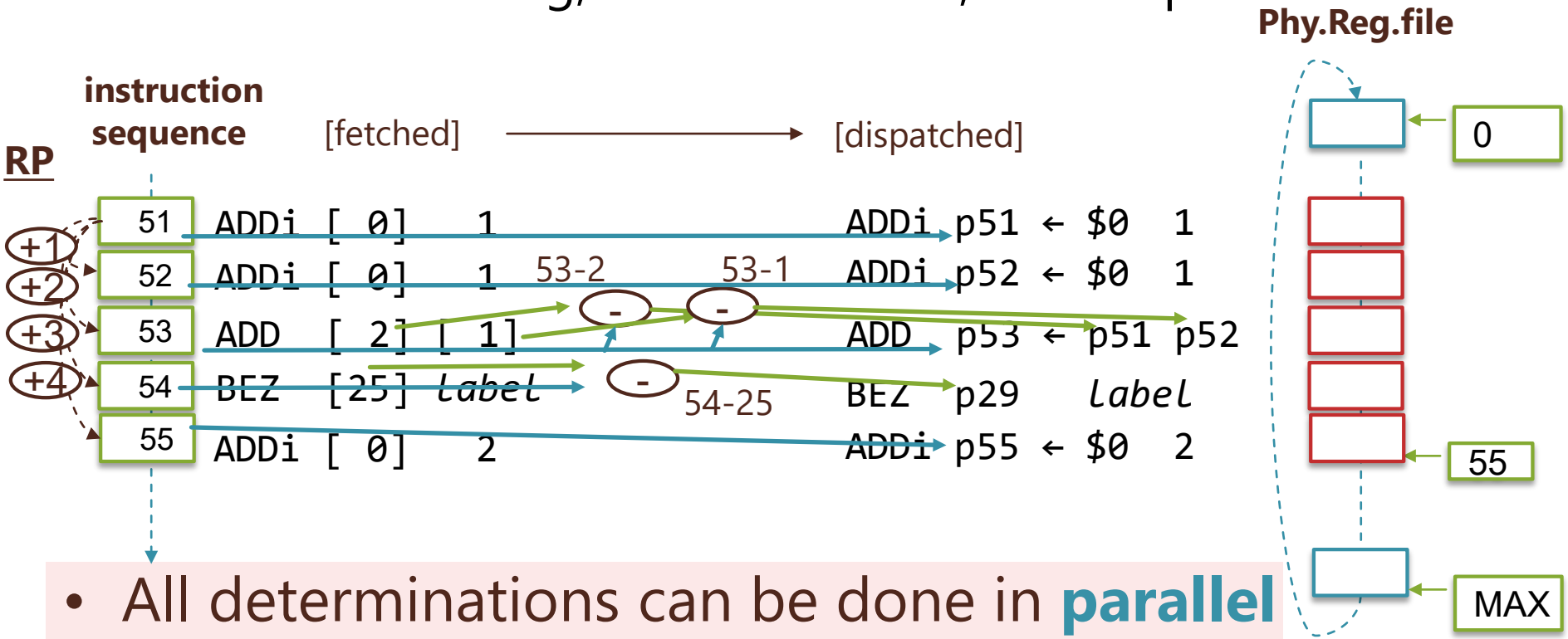
How the ISA can Simplify the Processor

- Mechanism of the operand determination
  - Conventionally processed by complex **register renaming**



How the ISA can Simplify the Processor

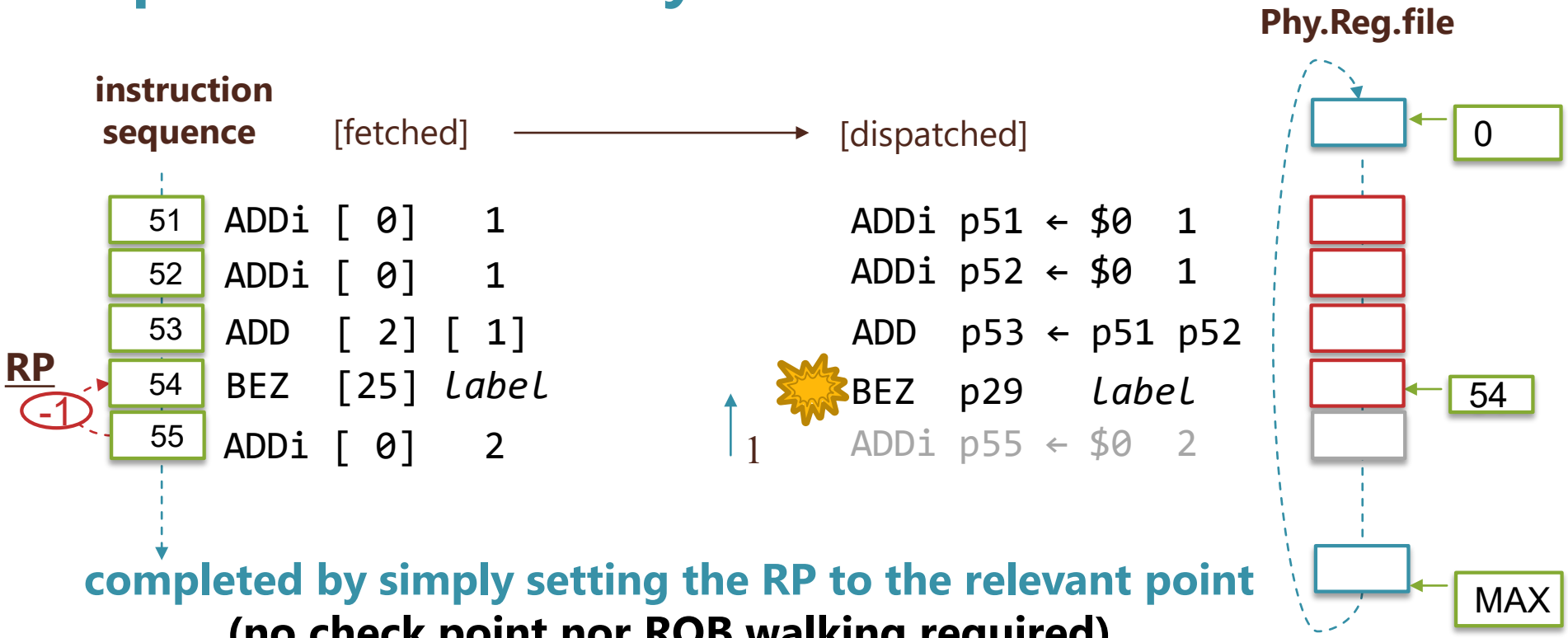
- The logic is much simpler than renaming
  - Unlike renaming; no table access, no complex state



- All determinations can be done in **parallel**
  - Improves the frontend scalability

How the ISA can Simplify the Processor

- The mechanism also **accelerates the miss prediction recovery**



completed by simply setting the RP to the relevant point
(no check point nor ROB walking required)

How do we write programs with distances?

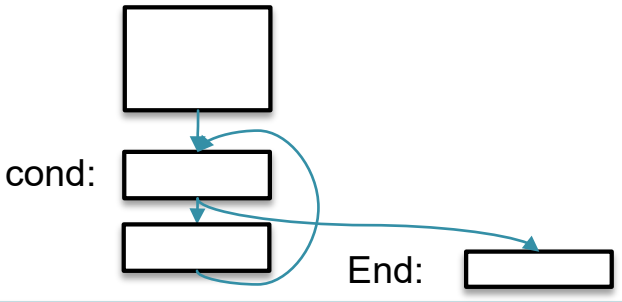
- Is it **possible to build a compiler?**
- Will the benefits surpass its **overheads?**

Determine the Viability of the architecture

Ex. Sequential block

```
ADDI  $0   1   #0 + 1 = 1
ADDI  $0   1   #0 + 1 = 1
ADD   [1]  [2] #1 + 1 = 2
ADD   [1]  [2] #2 + 1 = 3
ADD   [1]  [2] #3 + 2 = 5
ADD   [1]  [2] #5 + 3 = 8
ADD   [1]  [2] #8 + 5 = 13
ADD   [1]  [2]
```

Ex. Looped code




```
        RMOV   [5]
        RMOV   [5]
        ADDi   $0   0
        NOP
cond:   SLT    [2]  [3]
        BEZ    [1]  end:
        ST     [4]  [6]
        ADDi   [7]  4
        RMOV   [7]
        ADDi   [7]  1
        J      cond:
End:    SPADDi -4
```

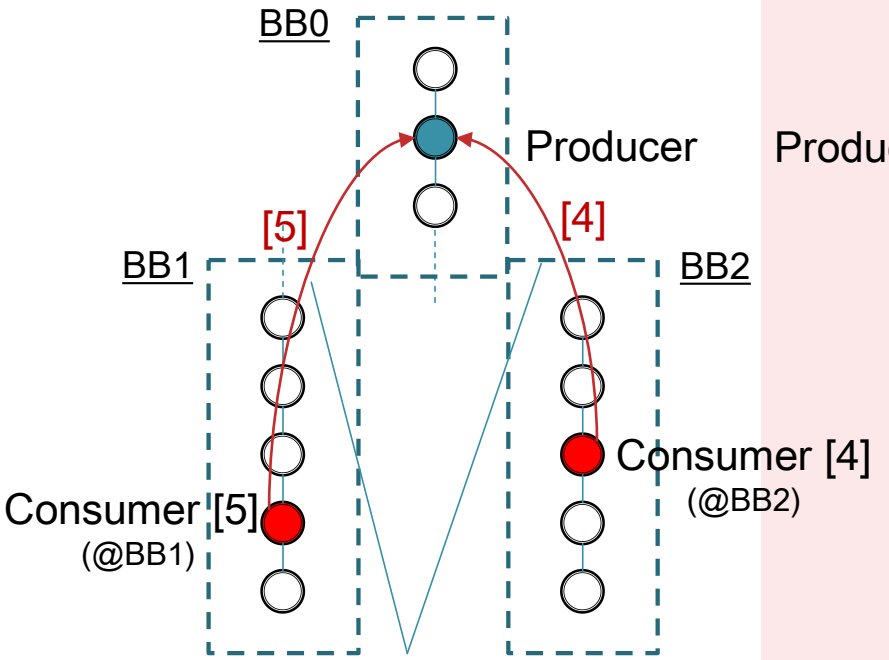
Compilation Algorithm for STRAIGHT

- **Any Single Static Assignment (SSA) form can be converted to STRAIGHT ISA code**
  - (so we can convert **LLVM-IR** to **STRAIGHT binary**)
- Key idea: inserting copy instructions (RMOVs) can satisfy the restriction derived from the ISA
  - **Repeater RMOVs** for long references
  - **Adjustment RMOVs** (explained later)

Concept of Distance Adjustment

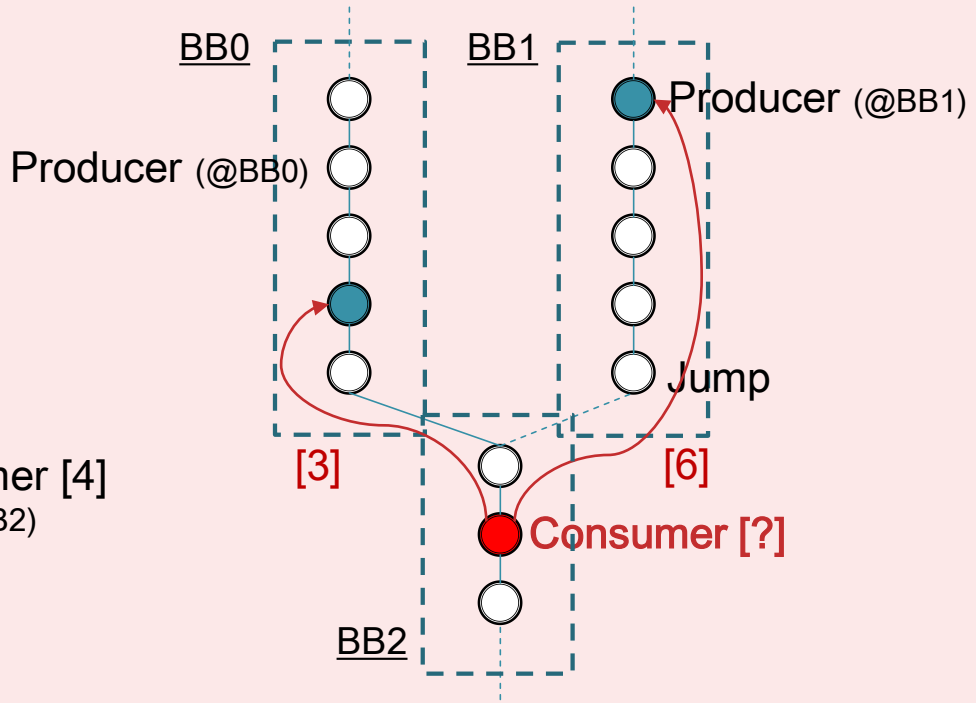
- All control flow structures are broken down into…

branching control flows



the path between the producer and each consumer is deterministic

merging control flows

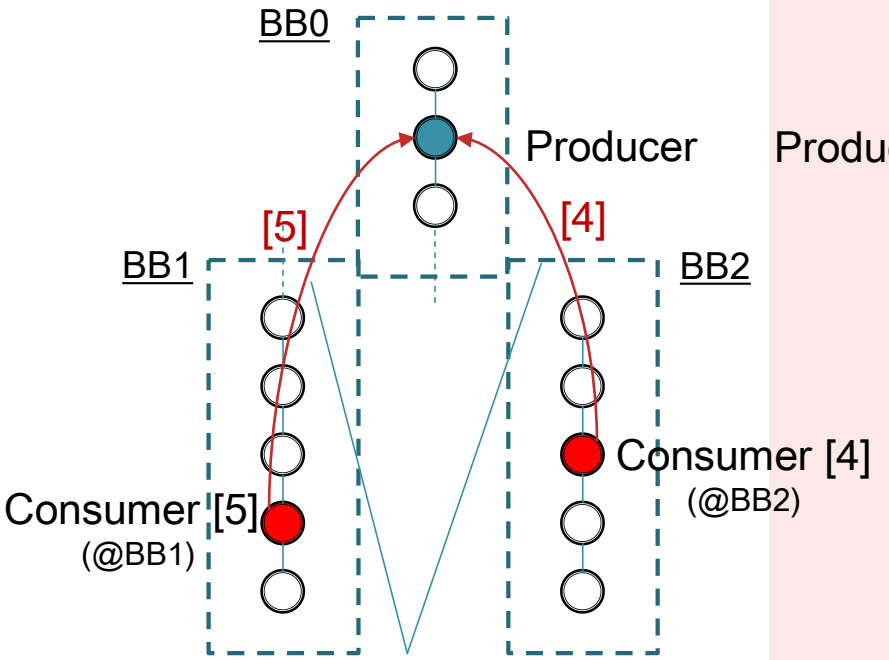


Multiple possible paths between the consumer and its producers

Concept of Distance Adjustment

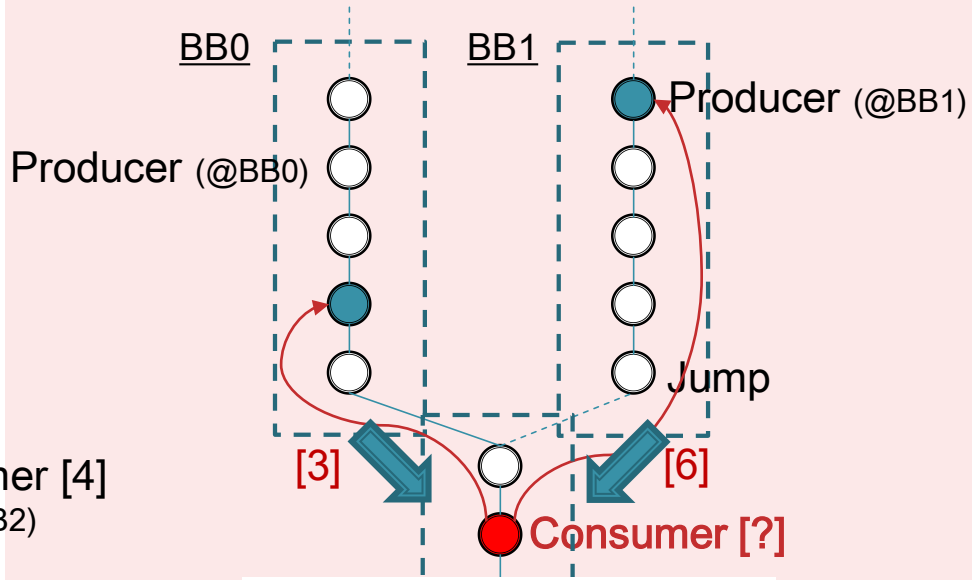
- All control flow structures are broken down into…

branching control flows



the path between the producer and each consumer is deterministic

merging control flows



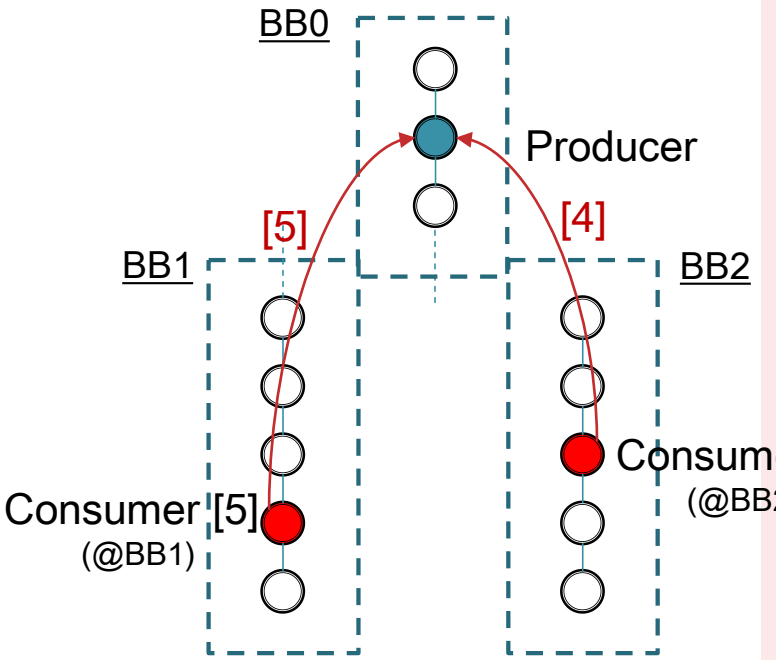
Distance varies dynamically depending on which path is taken

Multiple possible paths between the consumer and its producers

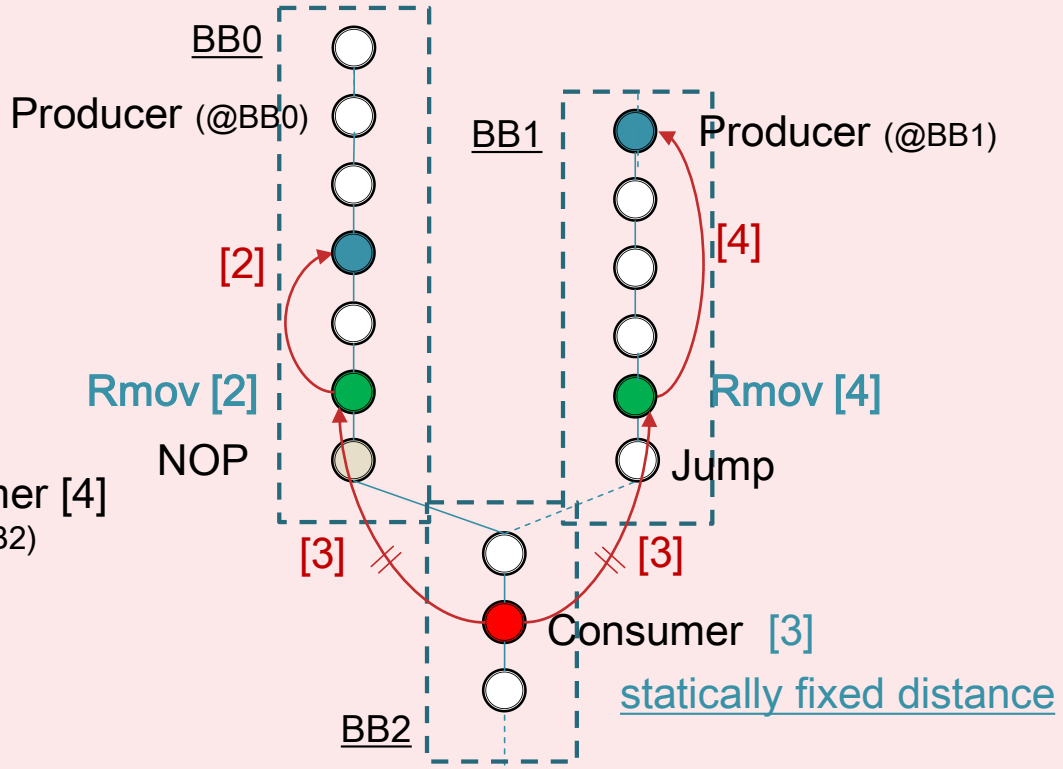
Breakdown of Control Flow

- All control flow structures are broken-down into…

branching control flows



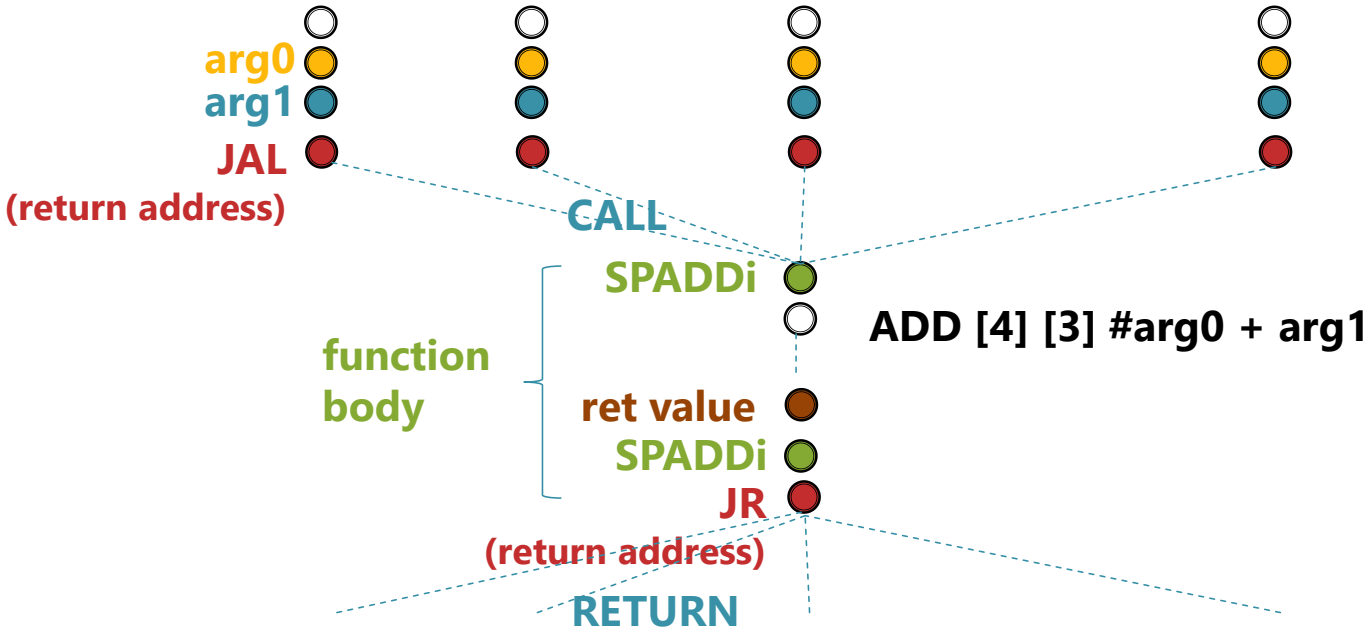
merging control flows



inserting an RMOV into the tail of each basic block can equalizes the distances

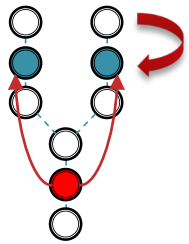
Calling convention

- Function calls can be achieved by defining the order of arguments and return values for each function at the compilation time

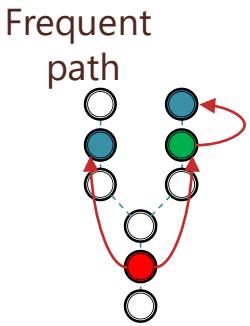


Optimization Technologies

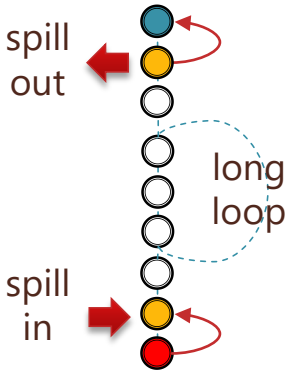
- Removing the redundant RMOV instructions
  - basic compilation algorithm tend to insert too much redundant RMOVs



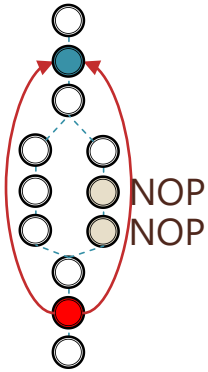
0-copy



half-copy



aggressive-spill



path-equalization

Pros and Cons of the Architecture

- ☺ Pros

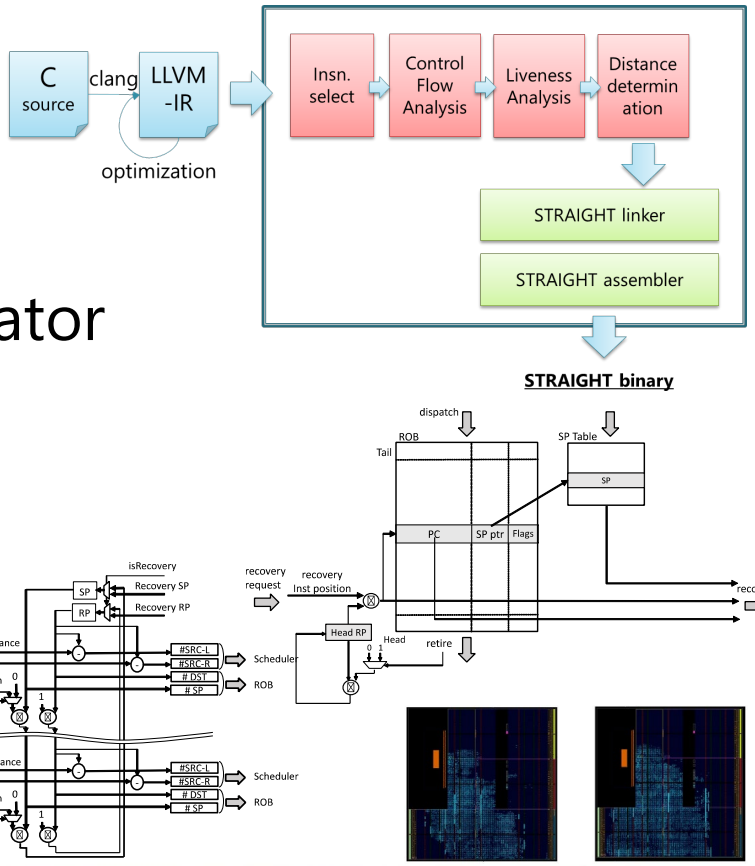
1. **Power efficiency**
   - ☺ power per instruction, ☺ removal of a hotspot
2. **Recovery performance**
   - ☺ branch miss penalty
3. **Scalability**
   - ☺ frontend width, ☺ROB → ☺ higher ILP extraction
4. **Spill reduction**
   - ☺ logical register space is highly extendable

- ☹ Cons

1. **Instruction count**

Preparation of the First Evaluation Environment

- Developed a real architectural framework of STRAIGHT for evaluating the actual effect
  - STRAIGHT ISA specification
  - STRAIGHT compiler (LLVM-backend)
  - STRAIGHT Cycle accurate simulator
  - STRAIGHT RTL description
  - OoO RISC-V(sim, RTL) (superscalar counterpart)



Preparation of the First Evaluation

- **Dhrystone / Core Mark were compiled**
  - Although these are simple benchmarks, Core Mark involves typical control-flow patterns of real applications

```
Function_Func_1 :
BB_0@Func_1 :
[    0] IMPLICIT_ARG_VALUE       # user: [    8], [   13]
[    1] IMPLICIT_ARG_VALUE       # user: [    8]
[    2] IMPLICIT_RET_ADDR        # user: [F   2], [F   2]
[    8] SUB      <  3>, <  2>     # [    0], [    1] # user: [   10]
[   10] BNZ      <  1>, BB_3@Func_1# [    8]
[   11] JMP      BB_1@Func_1
BB_1@Func_1 :
[   12] Global   Ch_1_Glob        # user: [   13]
[   13] STB      <  7>, <  1>, 0  # [    0], [   12]
[f  14] ADDi     <  0>, 1         # Z-REG            # user: [   18]
[F   2] RMOV     <  7>            # [    2]          # user: [   17]
[   15] JMP      BB_2@Func_1
BB_3@Func_1 :
[f   7] ADDi     <  0>, 0         # Z-REG            # user: [   18]
[F   2] RMOV     <  4>            # [    2]          # user: [   17]
[   16] JMP      BB_2@Func_1
BB_2@Func_1 :
[   17] PHI      <  2>, <  2>     # [F   2]or[F   2] # user: [   21]
[   18] PHI      <  3>, <  3>     # [f  14]or[f   7] # user: [Z  20]
[Z  20] RMOV     <  3>            # [   18]
[   21] JR       <  3>            # [   17]
~~~~
```

[Dhrystone (part)]

```
Function_core_list_find :
BB_0@core_list_find :
[    0] IMPLICIT_ARG_VALUE       # user: [   15], [   20], [F   0], [F   0]
[    1] IMPLICIT_ARG_VALUE       # user: [    9], [f  17]
[    2] IMPLICIT_RET_ADDR        # user: [    6]
[    3] SPADDi  -4               # user: [    6]
[    6] SpillST <  2>, <  1>, 0  # [    2], [    3]
[    9] LDH      <  4>, 2        # [    1]          # user: [   11], [F   9]
[   10] ADDi     <  0>, -1       # Z-REG            # user: [   11]
[   11] SLT      <  1>, <  2>     # [   10], [    9] # user: [   12]
[   12] BNZ      <  1>, BB_3@core_list_find# [   11]
[   13] JMP      BB_1@core_list_find
BB_1@core_list_find :
[   15] BEZ      < 10>, BB_11@core_list_find# [    0]
[   16] JMP      BB_2@core_list_find
BB_2@core_list_find :
[F   0] RMOV     < 12>            # [    0]          # user: [   37]
[f  17] LDH      < 12>, 0         # [    1]          # user: [   36]
[   18] JMP      BB_7@core_list_find
BB_3@core_list_find :
[   20] BEZ      <  9>, BB_13@core_list_find# [    0]
[   21] JMP      BB_4@core_list_find
BB_4@core_list_find :
[F   0] RMOV     < 11>            # [    0]          # user: [   24]
[F   9] RMOV     <  7>            # [    9]          # user: [   23]
[   22] JMP      BB_5@core_list_find
BB_5@core_list_find :
~~~~
```

[Core Mark (part)]

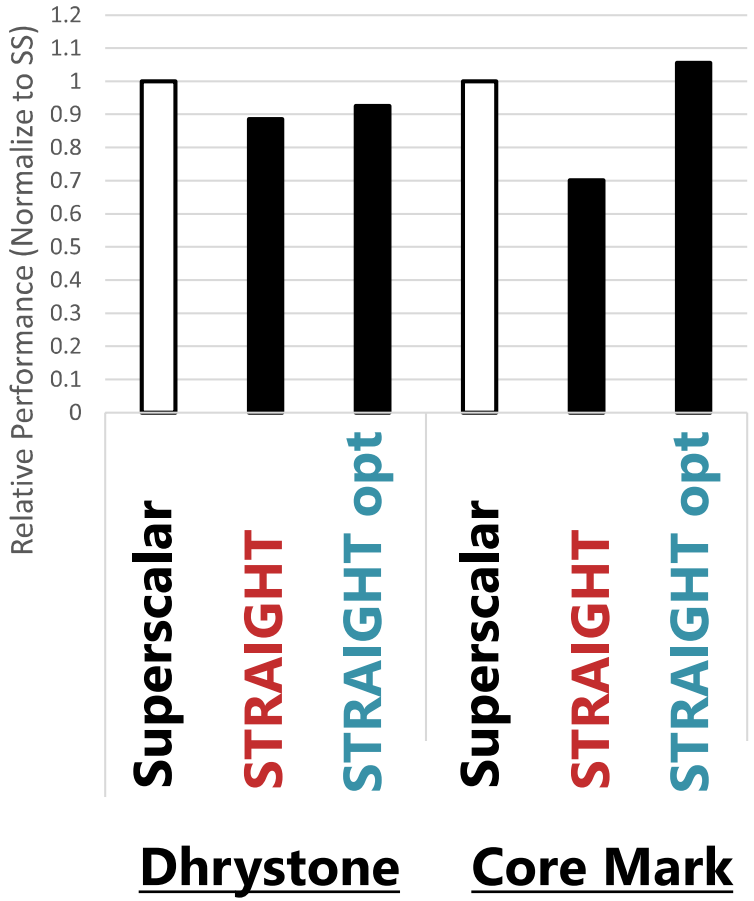
Processor Parameters

- Same sizes are set for the both architecture
  - Not optimal for STRAIGHT;
    Just for the simple comparison
  - **Max distance: 32**

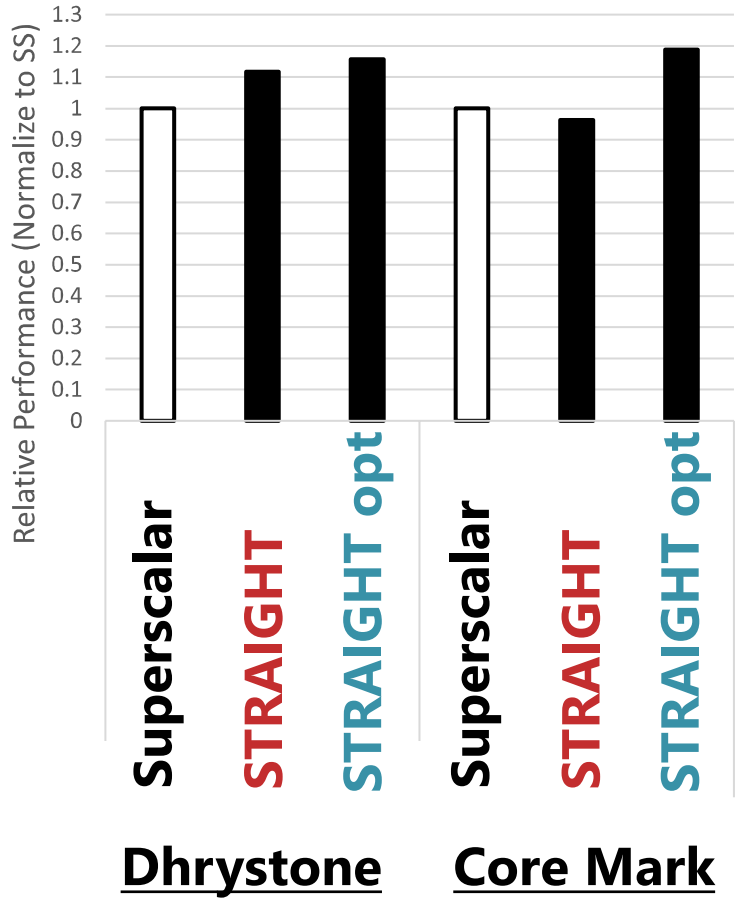
	2way		4way	
	SS	STRAIGHT	SS	STRAIGHT
ISA	RV32IM	STRAIGHT	RV32IM	STRAIGHT
Frontend Width	2 inst. / cycle		6 inst. /cycle	
Frontend Latency	8 cycle	6 cycle	8cycle	6cycle
RF, ROB	96 entries, 64 entries		256 entries, 224 entries	
Scheduler	16 entries, 2 issues / cycle		96 entries, 4 issues / cycle	
Caches	32 KB I1, 32KB D1, 256KB L2		+ 2MB L3	

The performance evaluation

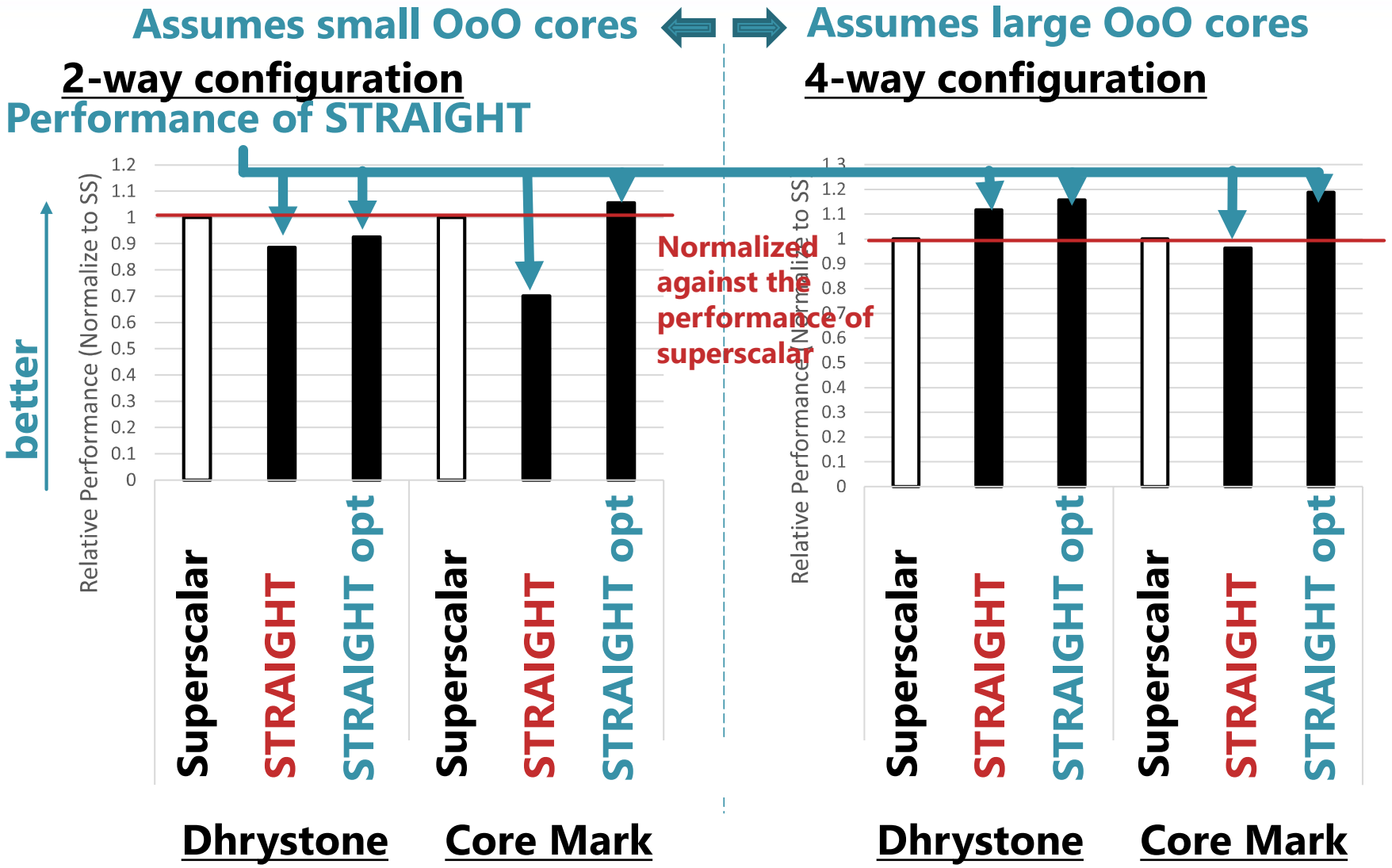
2-way configuration



4-way configuration

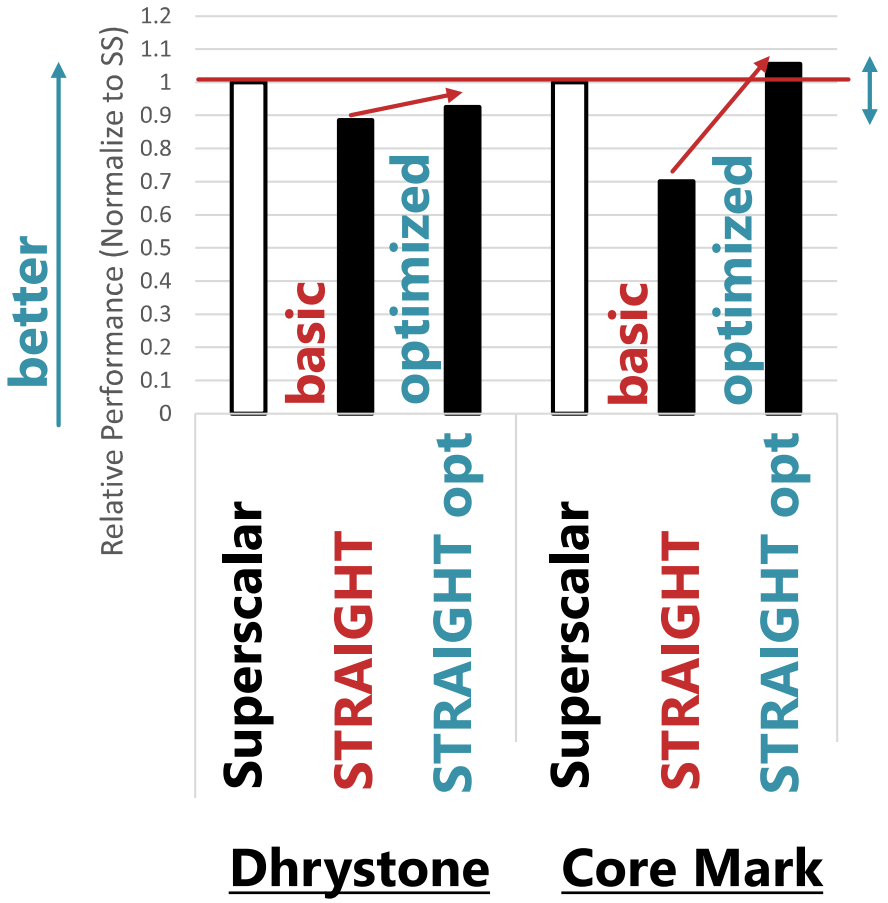


The performance evaluation

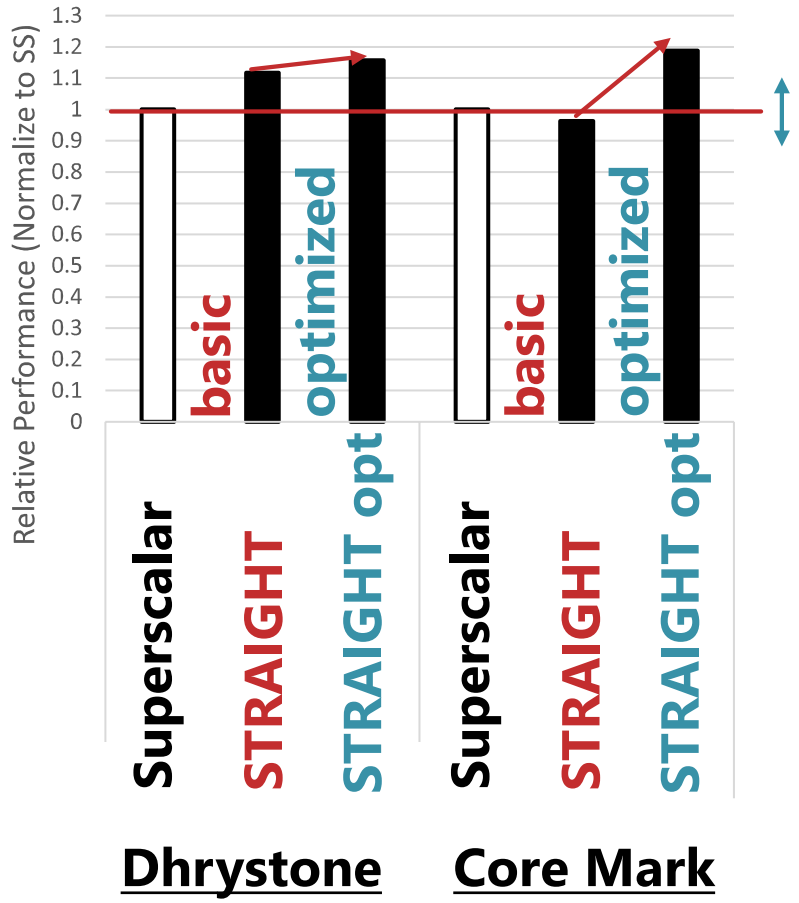


The performance evaluation

2-way configuration

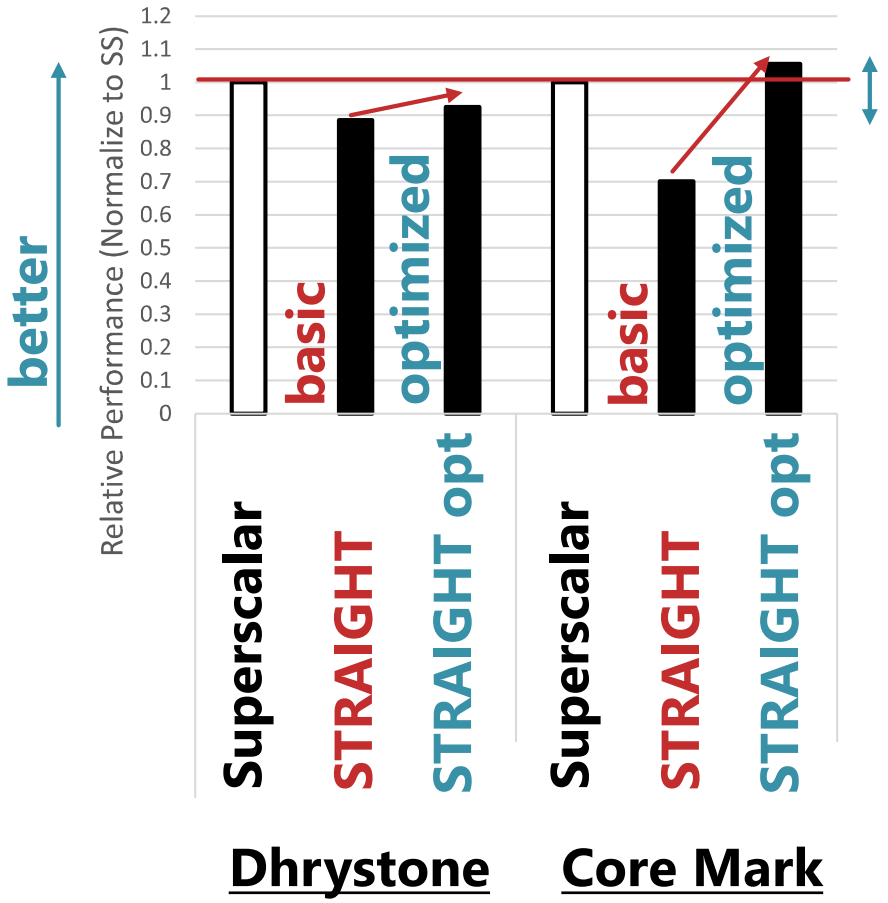


4-way configuration

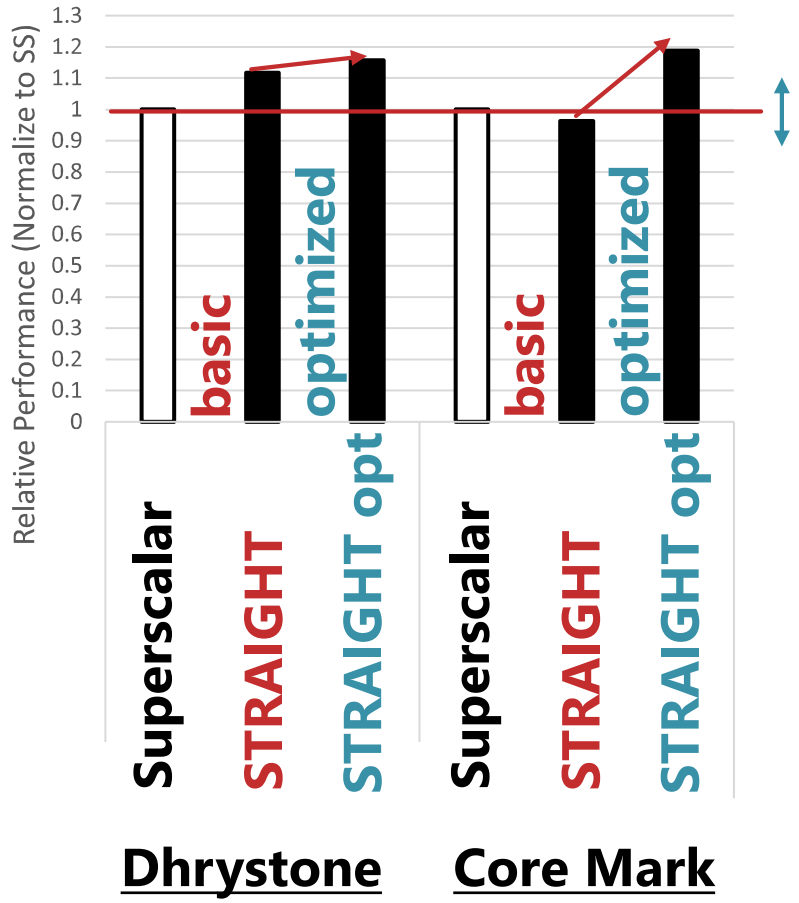


The performance evaluation

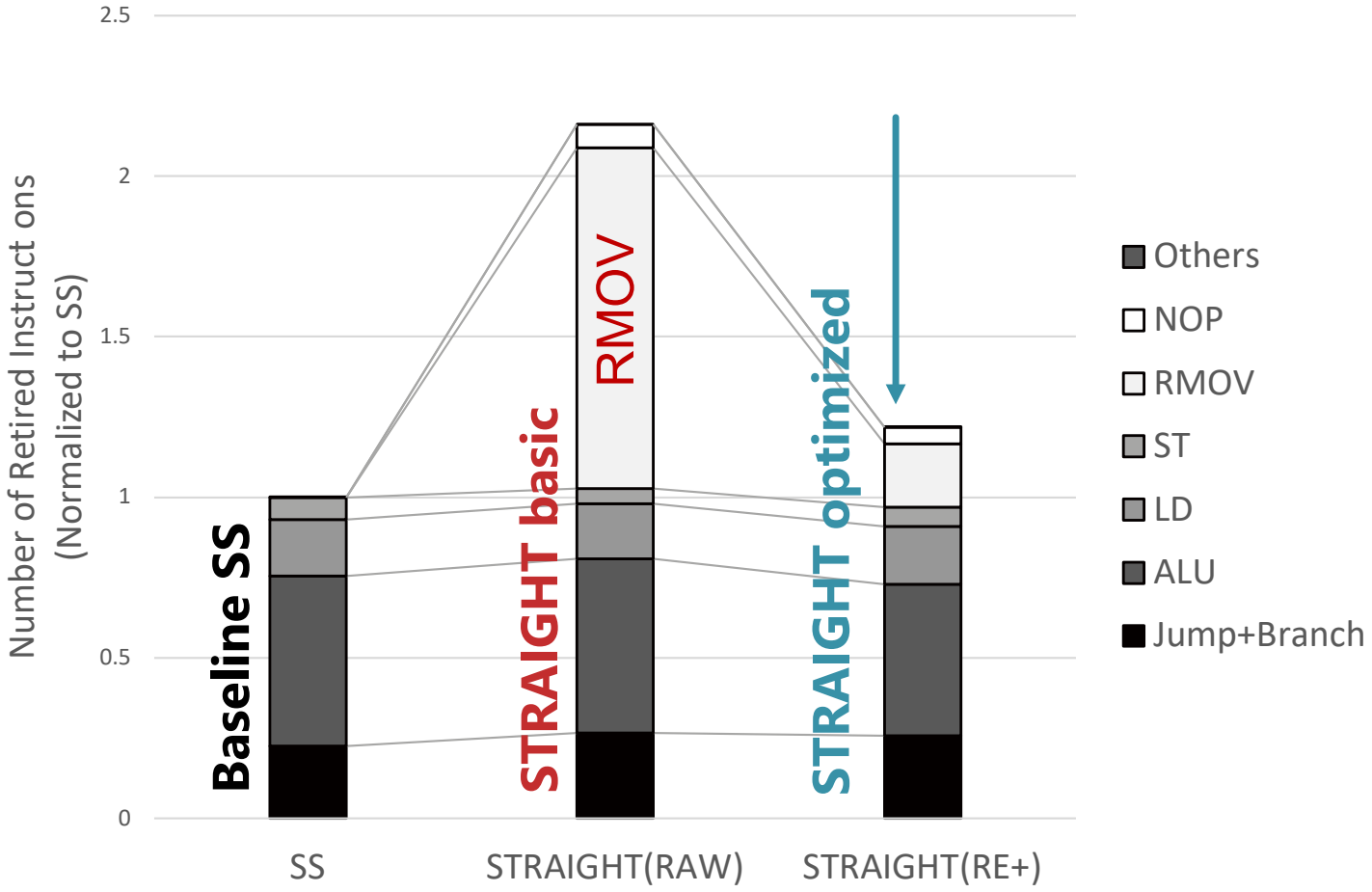
2-way configuration



4-way configuration

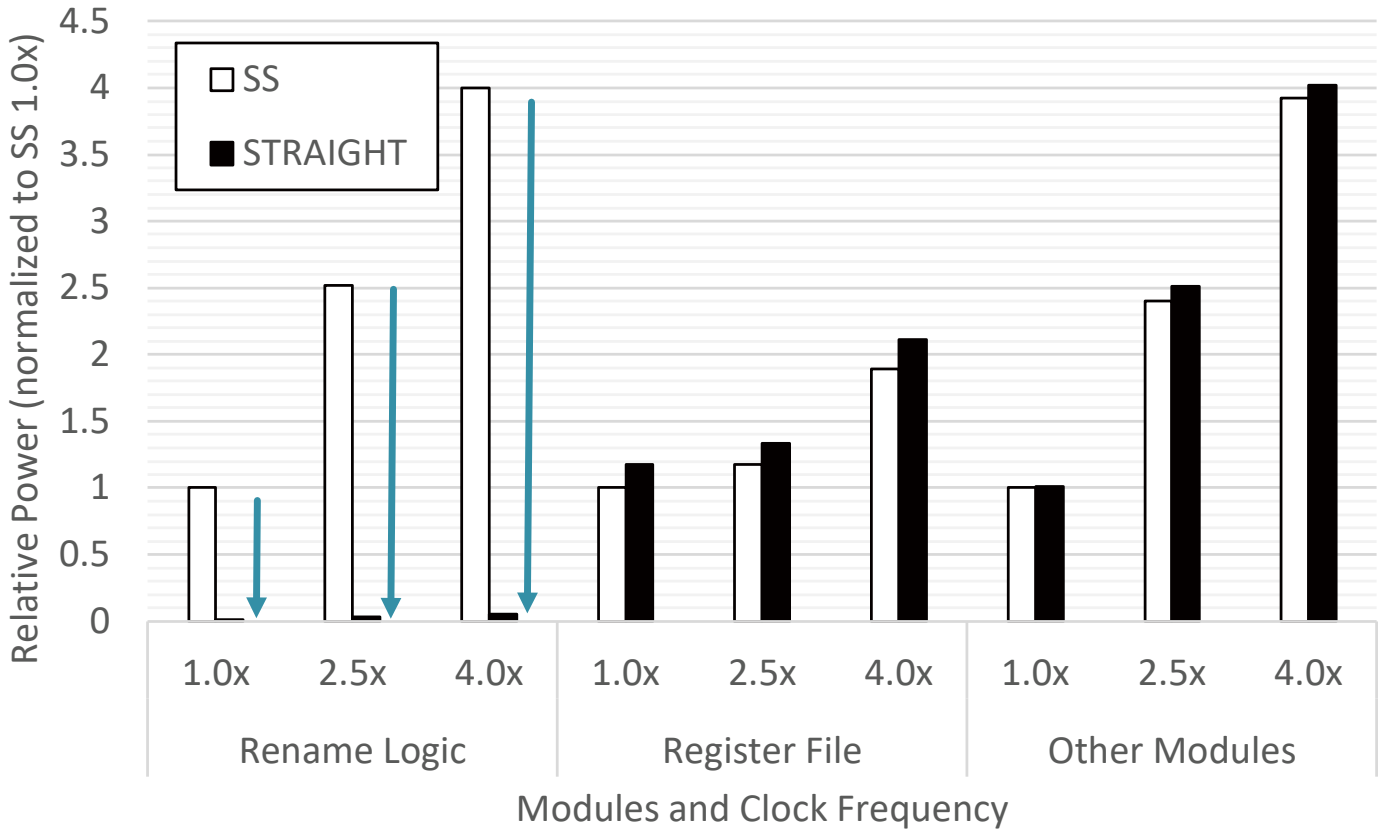


Fraction of the Retired Instruction Type



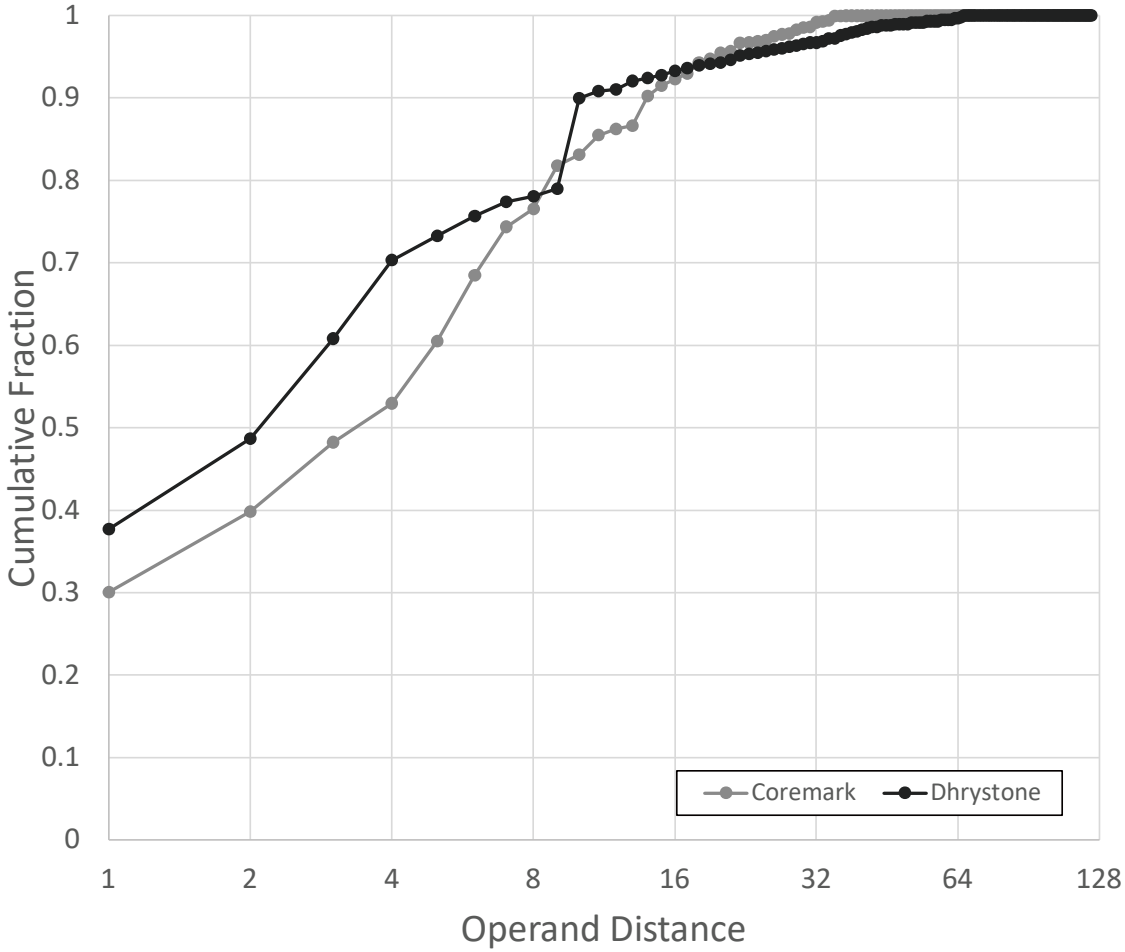
Core Mark

RTL Power Analysis



Power for the operand determination is almost removed

Distance distribution



Summary

- Novel effective OoO architecture STRAIGHT
- Expressing source operand by distance
  ⇒ makes the OoO processor simple/scalable/rapid recovery
- Code can be actually compiled from the SSA form
- The first evaluation
  - The performance is comparable to the same-sized superscalar
    - While it preserves the advantages of power and scalability

まとめ

2018/11/12

マイクロプロセッサ・アーキテクチャ

- 最近のプロセッサ設計傾向
  - 微細化の効果・速度鈍化の影響
  - 使える向上要素をバランス良く，多角的に
  - ILP技術研究の需要とオープンコアCPUの需要
- CPUコア性能を向上させる技術
  - 高性能コアのマイクロアーキテクチャ
- 研究中のアーキテクチャの紹介
  - コンパイラ支援によりリネームレスooo実行を実現
  - オープン化して公開予定