高性能計算基盤

自習 APDX06:区間再利用と事前実行

http://archlab.naist.jp/Lectures/ARCH/x06/apdx06j.pdf

ナレータ VOICEVOX:もち子(cv 明日葉よもぎ)

An Example: Factorial

```
main() {
  return F(4) + F(3) + F(5);  // result = 150
}

F(N) {
  if (N == 1) return 1;
  else        return N * F(N-1);
}
```

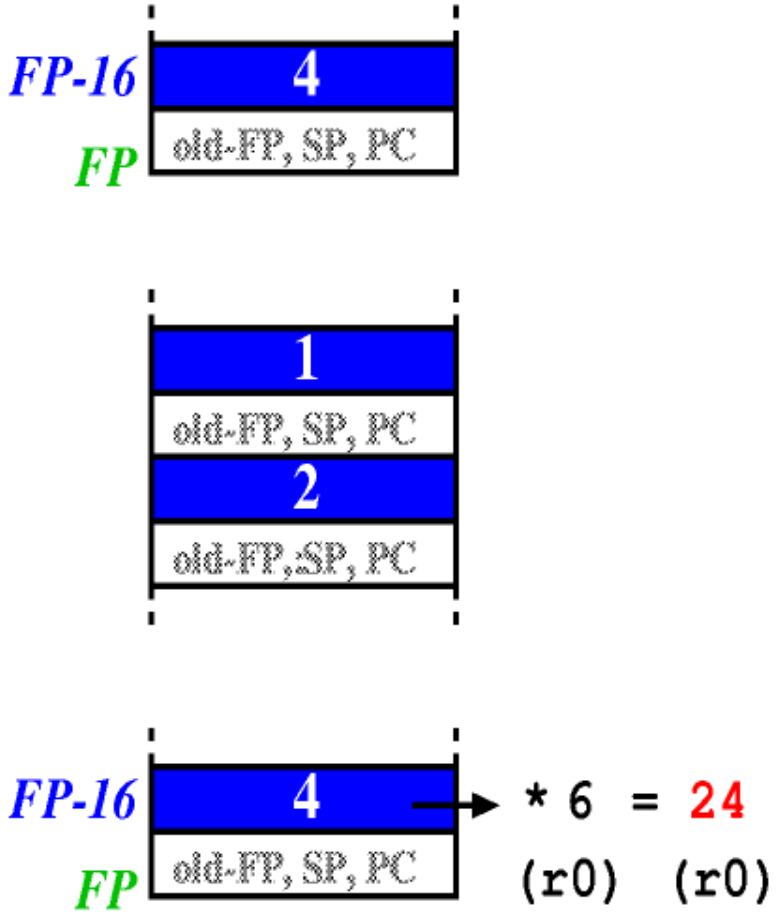
Main() requires 3 + 2 + 4 multiplication.

Can the computer get 150 with only 4 multiplication?

Normal Compilation

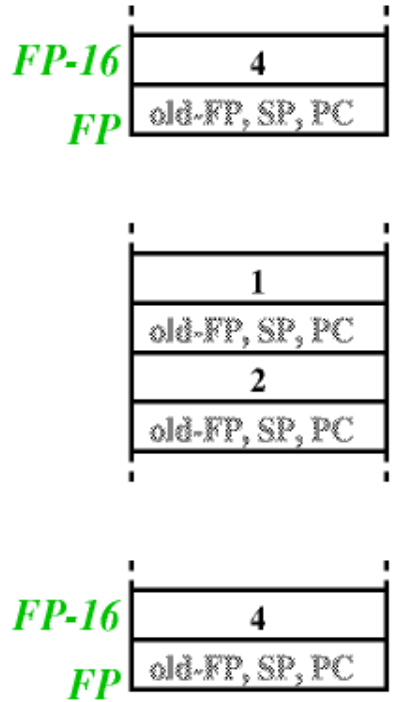
- Stack frame for temporal values
- R0 for argument (4) and return value (24)

```
F:   store  old-FP,SP,PC⇒[FP--]
     store  r0⇒[FP-16]
     branch if r0==1
-------------------------------------
     decr   r0
     call   F
-------------------------------------
     load   [FP-16]⇒r1
     mult   r1*r0  ⇒r0
end: load   [FP]⇒FP,SP,PC
```



Essential Input and Output


```
F:   store  old-FP,SP,PC⇒[FP--]
     store  r0⇒[FP-16]
     branch if r0==1
-------------------------------
     decr   r0
     call   F
-------------------------------
     load   [FP-16]⇒r1
     mult   r1*r0   ⇒r0
end: load   [FP]⇒FP,SP,PC
```



- If all sources (address and value) are the same as before, we can reuse F.
- If the address < FP, the variable is local.
- Excluding the locals (old-FP, SP, PC, [FP]) enables relocatable reuse.

Relocatable Memoization

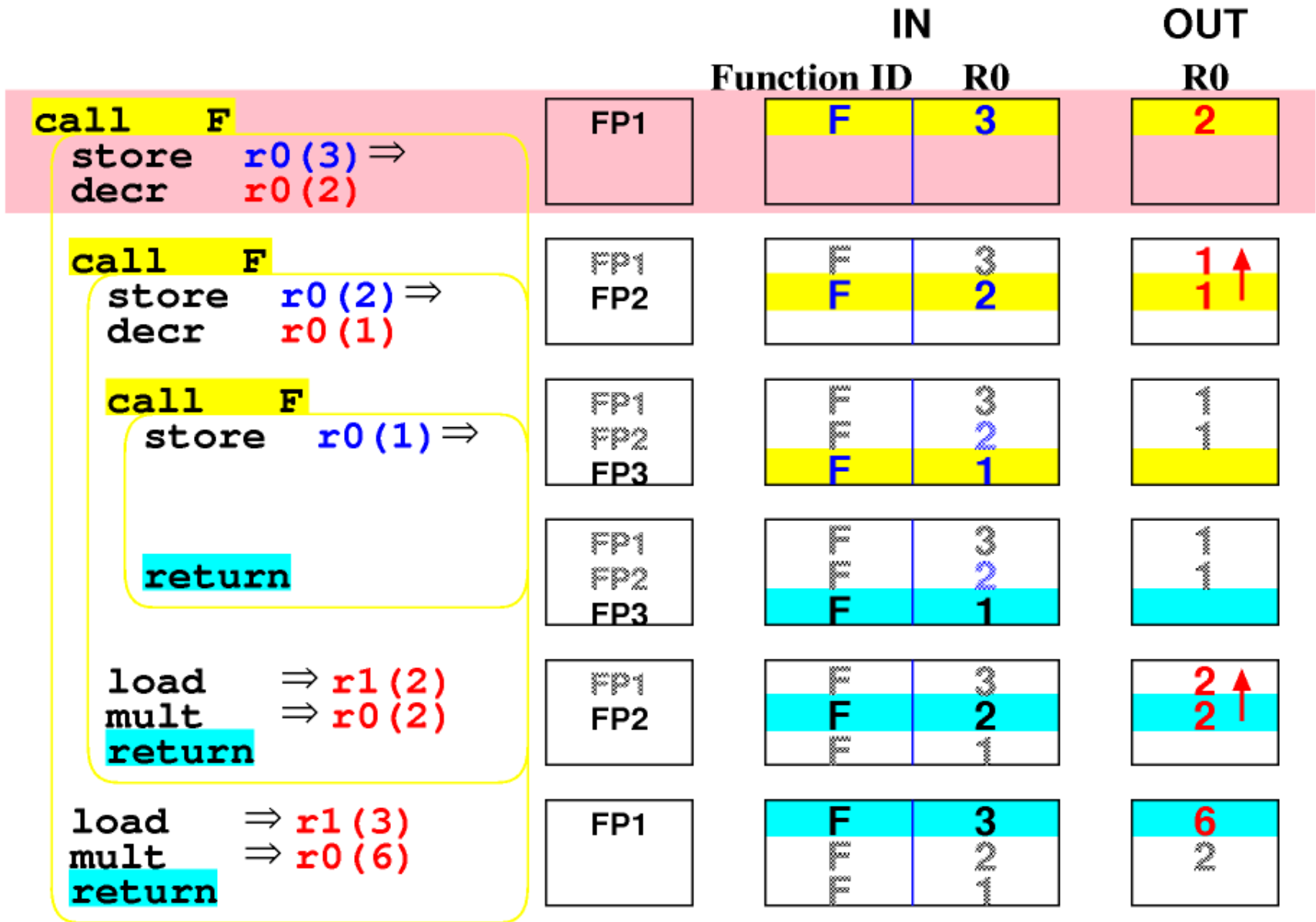
```
call   F

store  r0 ⇒ --
decr   r0 ⇒ r0
load   -- ⇒ r1
mult   -- ⇒ r0
return
```

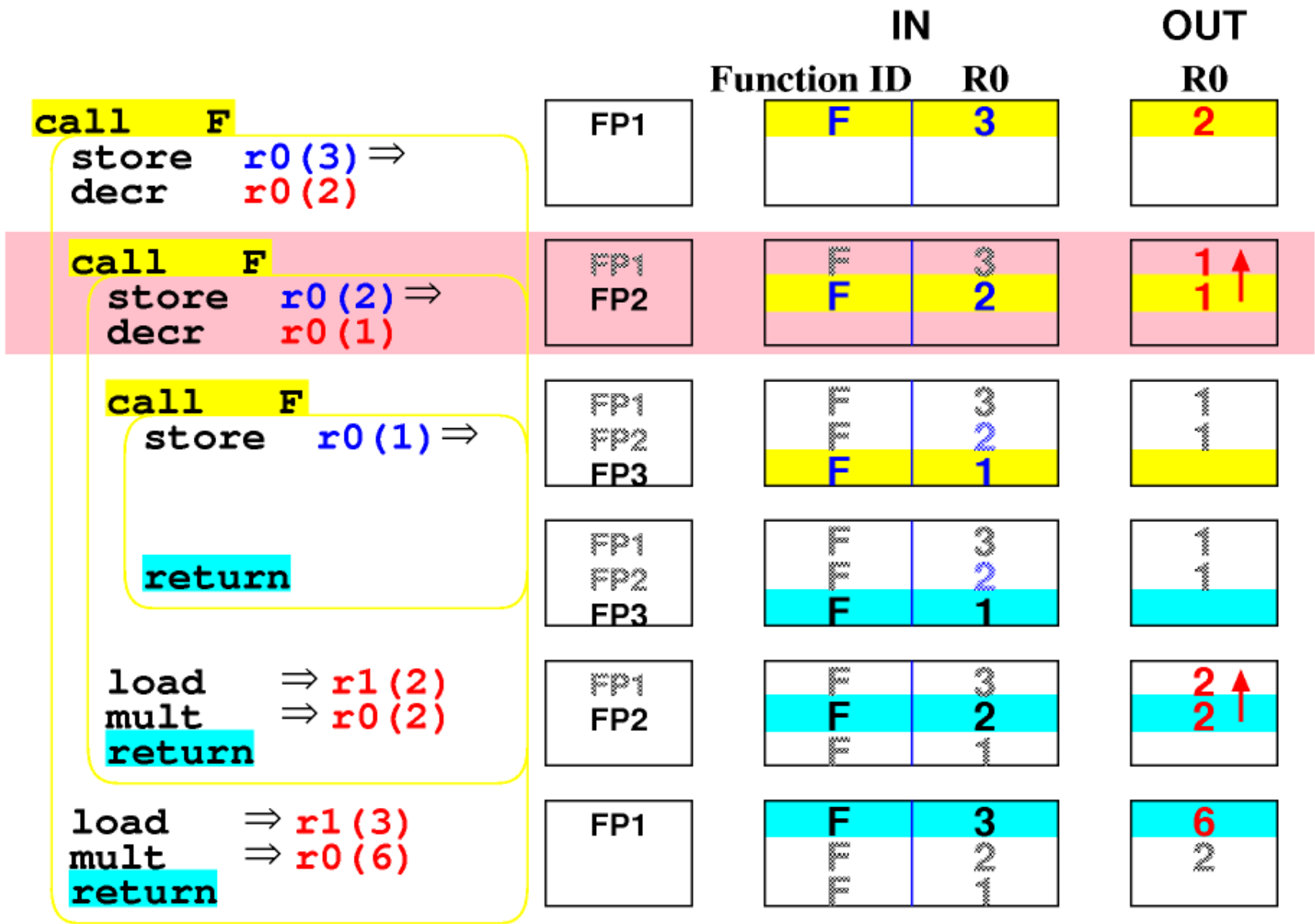
- R0 is read then written. ⇒ input and output
- R1 is written first. ⇒ output

- If R1 is local-reg like SPARC, can exclude R1.

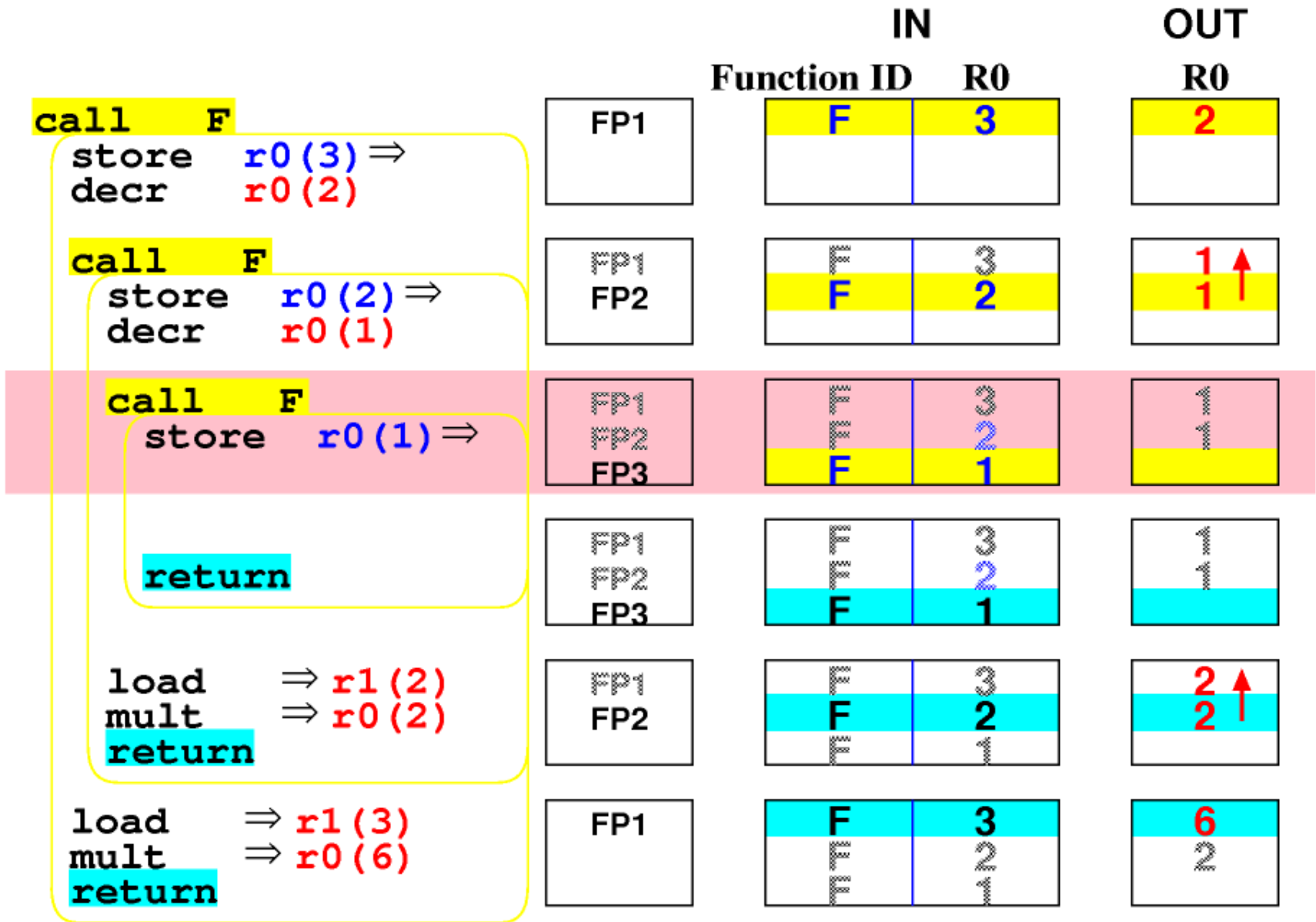
Multilevel Memoization



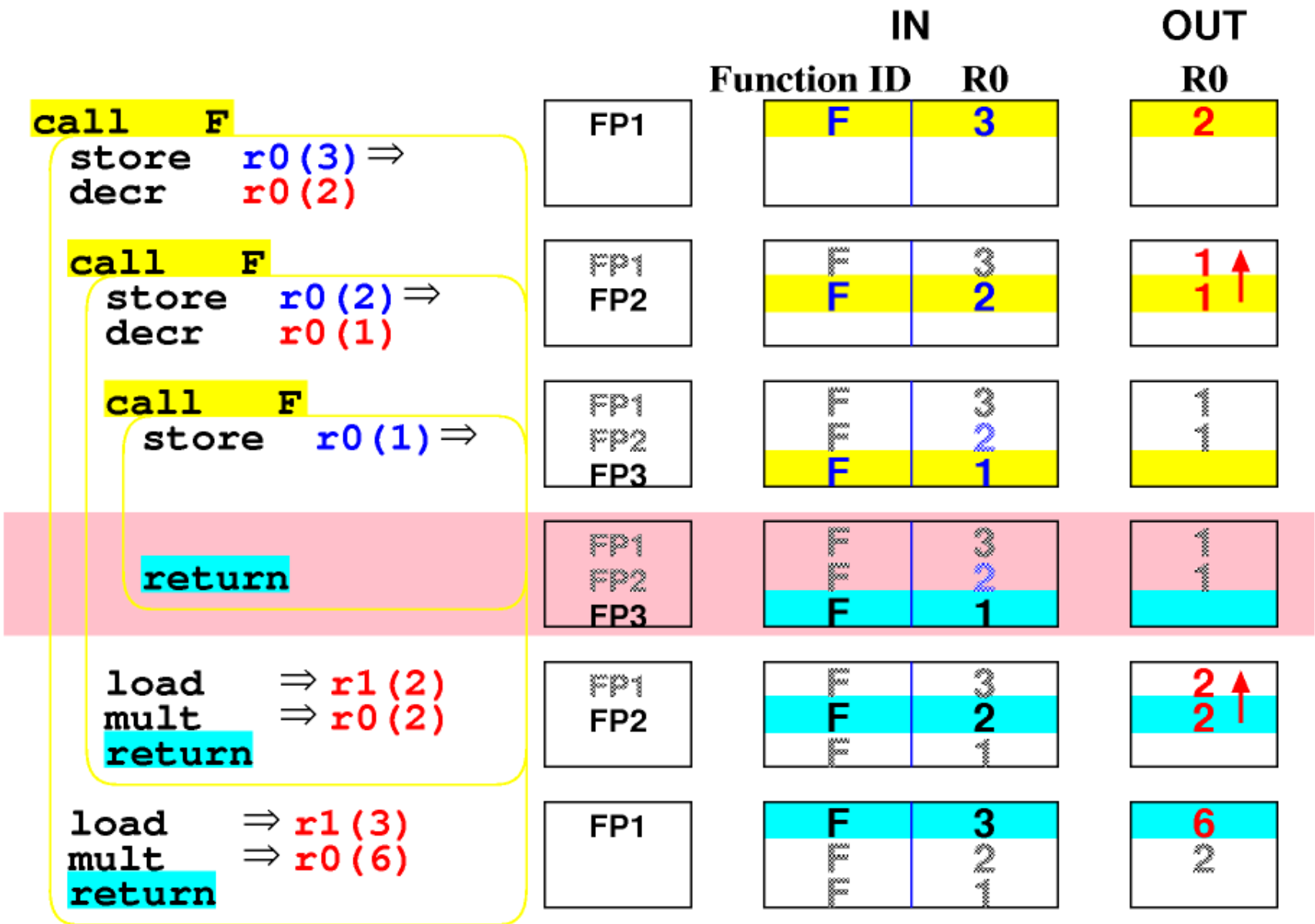
Multilevel Memoization



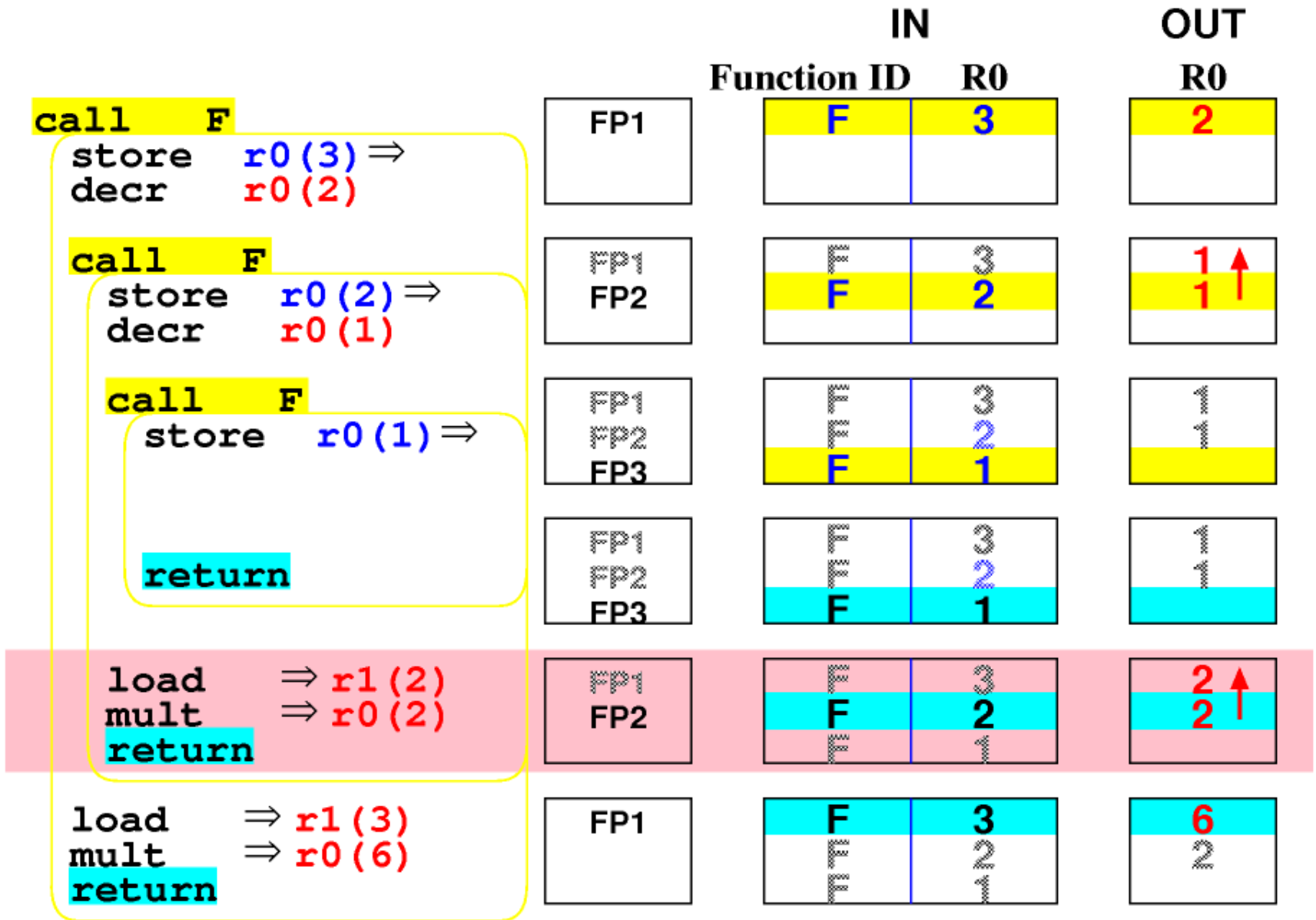
Multilevel Memoization



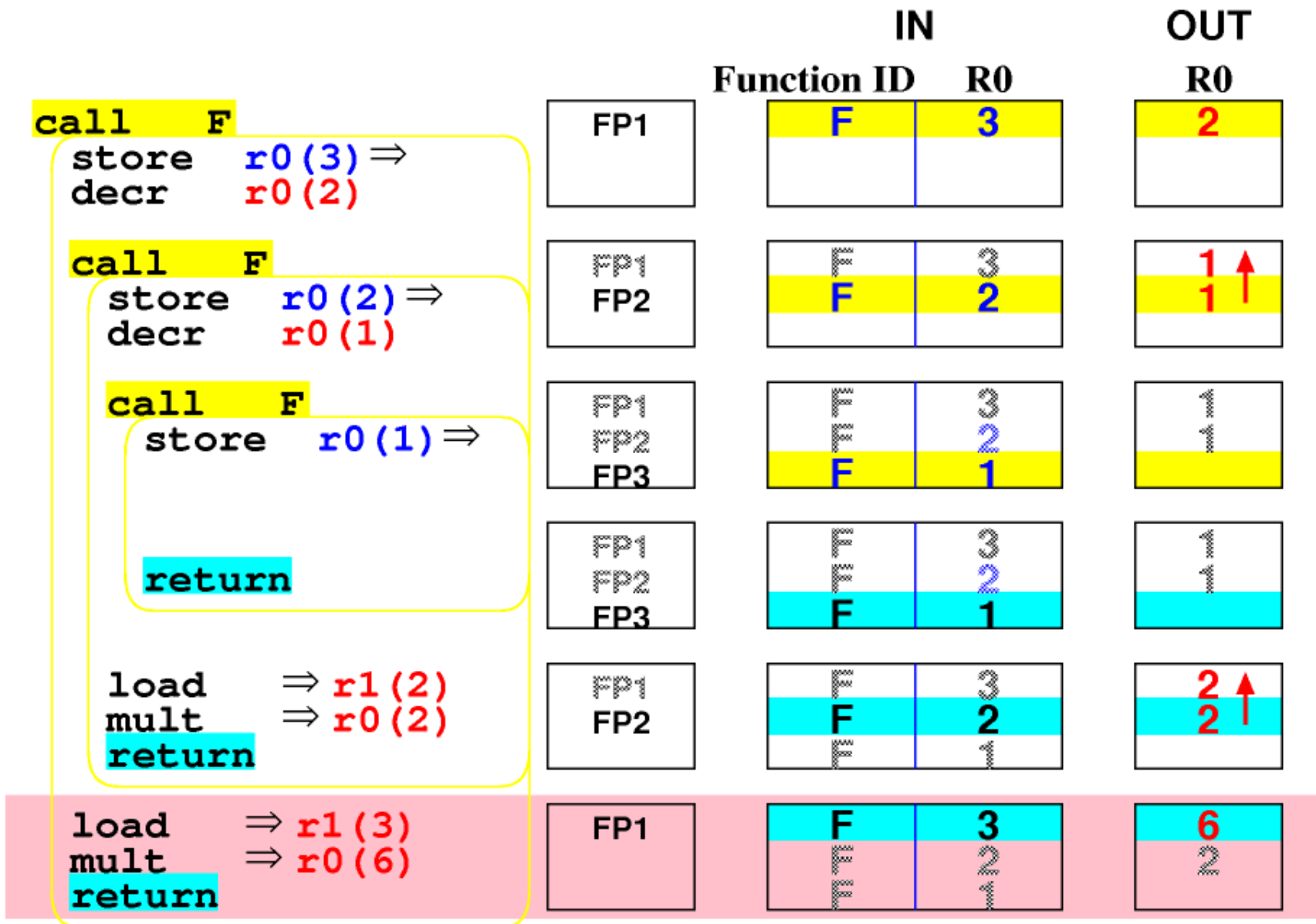
Multilevel Memoization



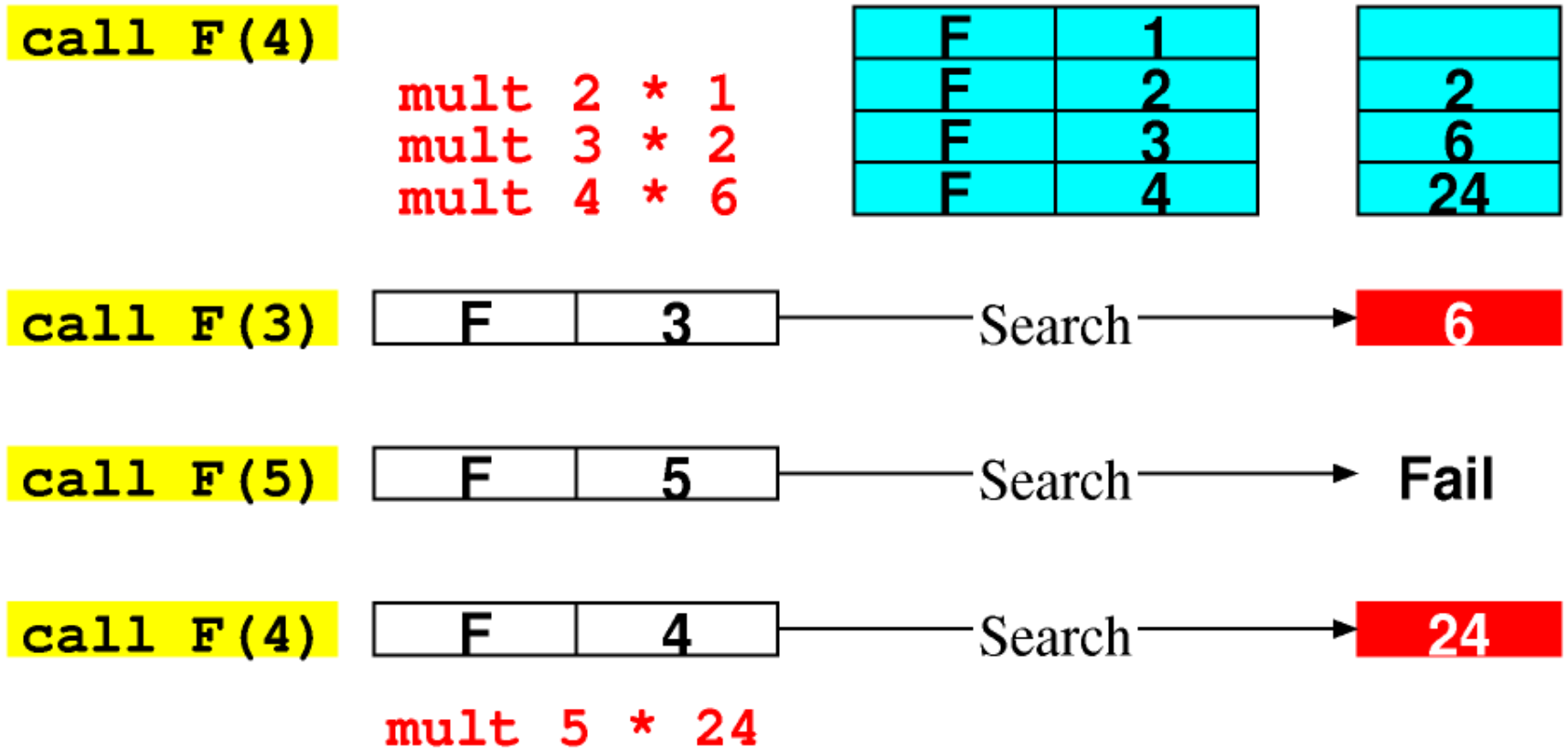
Multilevel Memoization



Multilevel Memoization



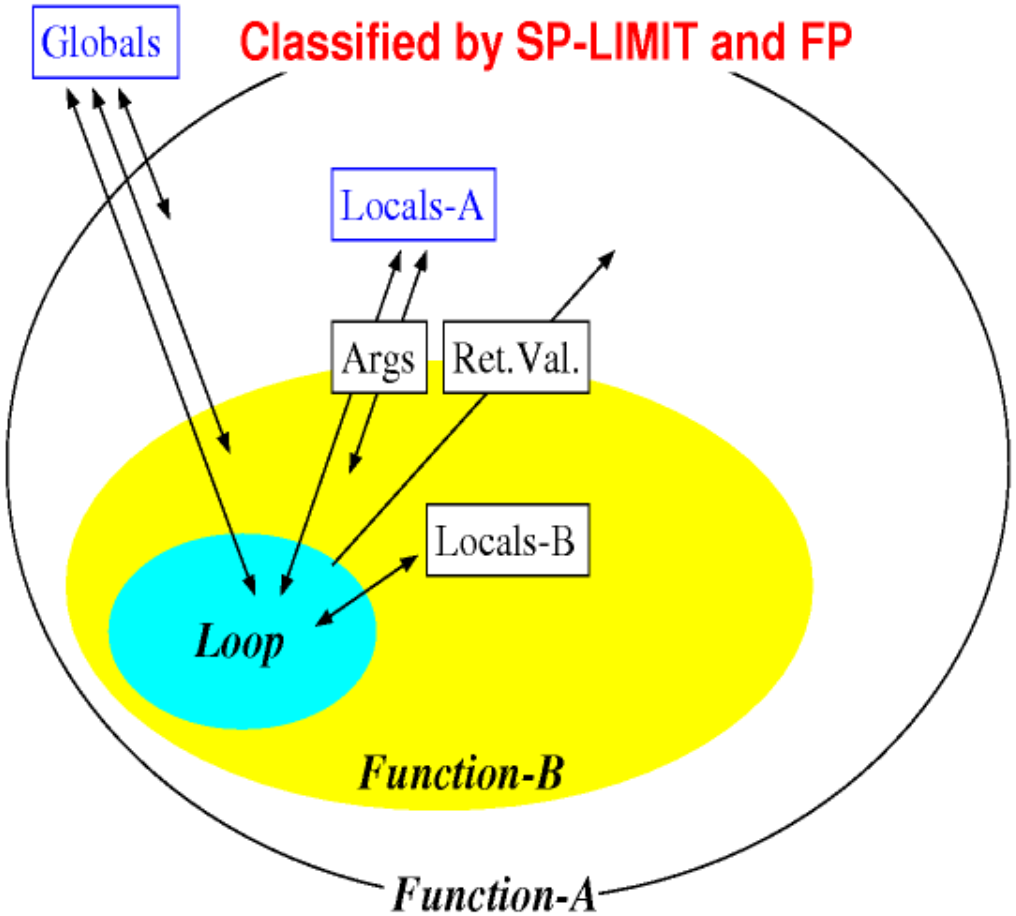
Reuse of Previous Results

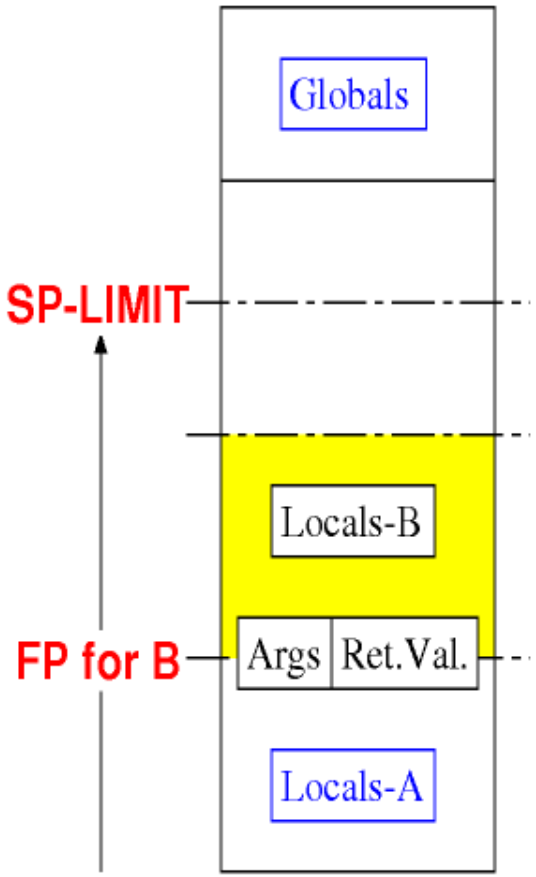


Can the computer get 150 with only 4 mult?

Extension for General Programs

In general, globals and callers' locals should be memoized.
SP-LIMIT provided by OS helps the classification.





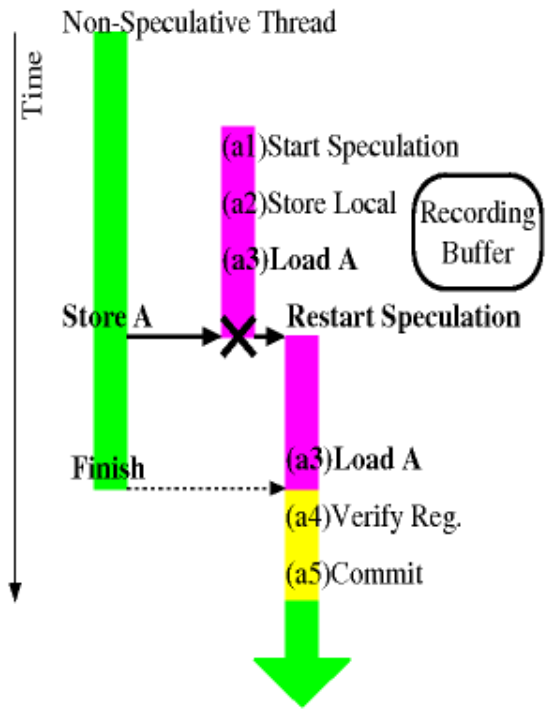
Another Example: Puzzle in Stanford Benchmark


```
T(j): loop  if (F(i, j)) {
              k = P(i, j);
              if (T(k) || k == 0 ) return true;
              else R(i, j);
            }
      return false;
```

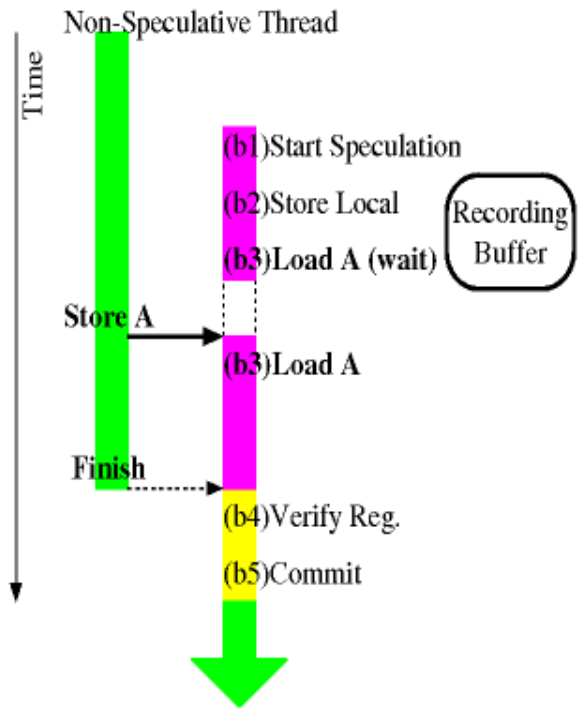
- No-reuse: 40 million instructions
- 256K table: 6 million instructions

- Average of 400 instr. are eliminated at once.

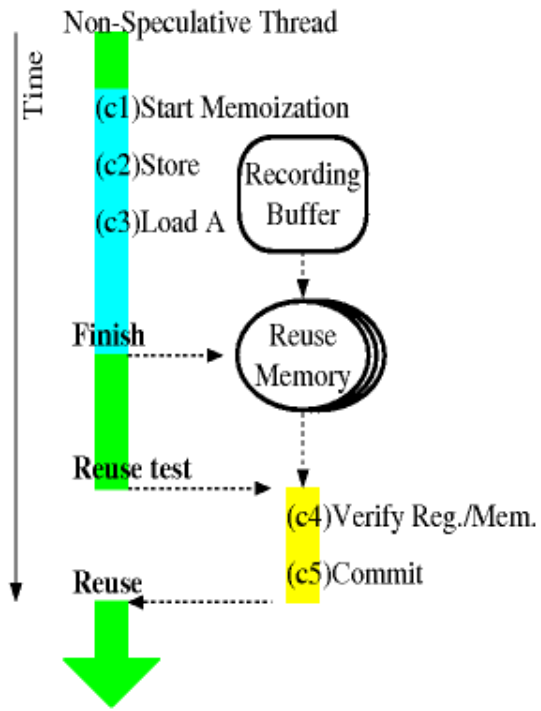
一般的な投機的マルチスレッディングとの比較



(a) 無効化するスレッド投機



(b) 待たせるスレッド投機



(c) 区間再利用

実行モデル「入力履歴⇒予測⇒投機実行⇒記録⇒再利用」

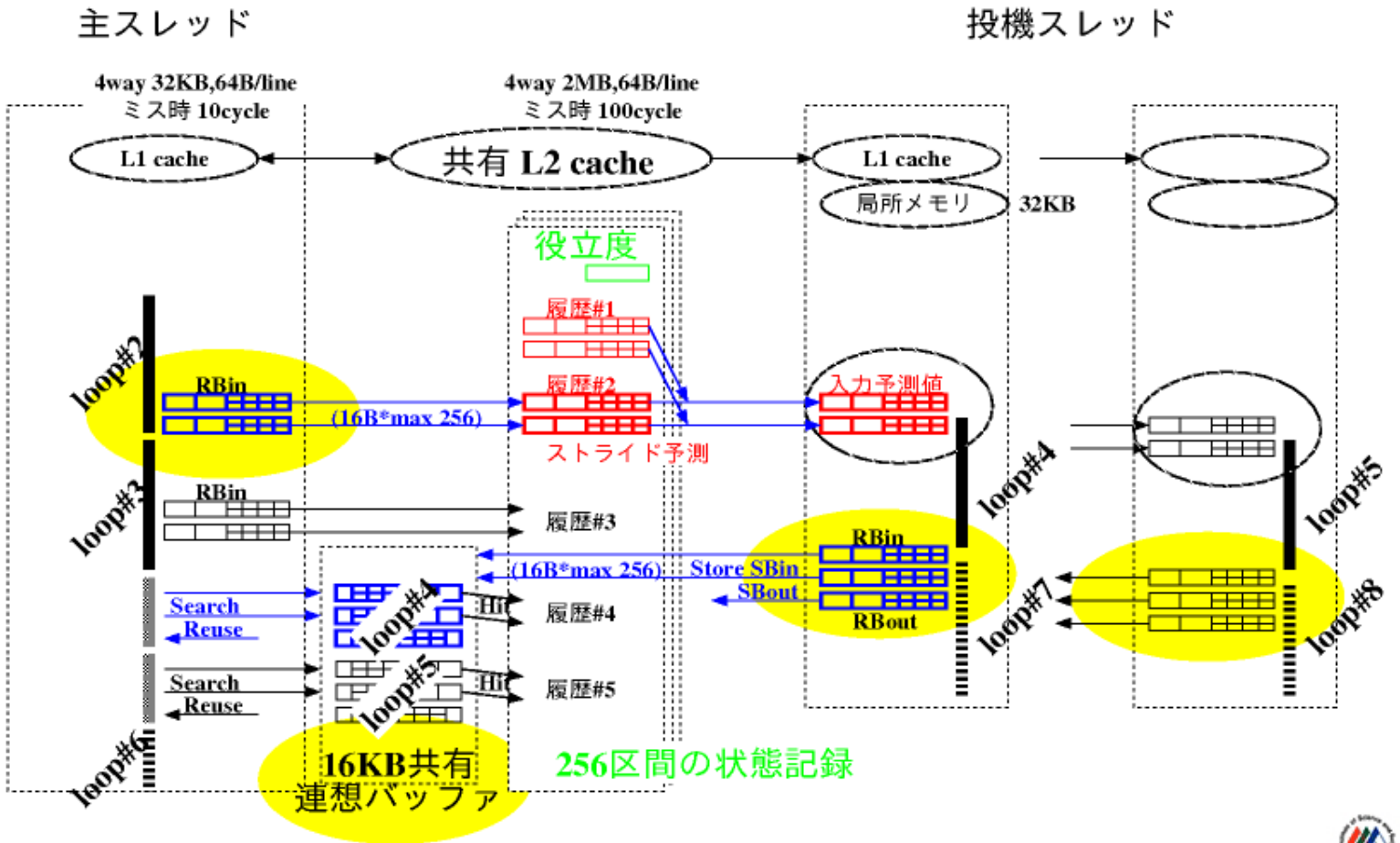
各スレッドごとに多重命令区間の入出力をリストに記録

► １６バイト×６４レコード×６レベル（６ＫＢ）

区間終了時に木構造として共有連想バッファ（１６ＫＢ）に集積

実行／再利用における入力履歴から，未来の入力を予測し事前実行

同一入力の探索と出力の再利用



NAIST

Example: strlen(strings)

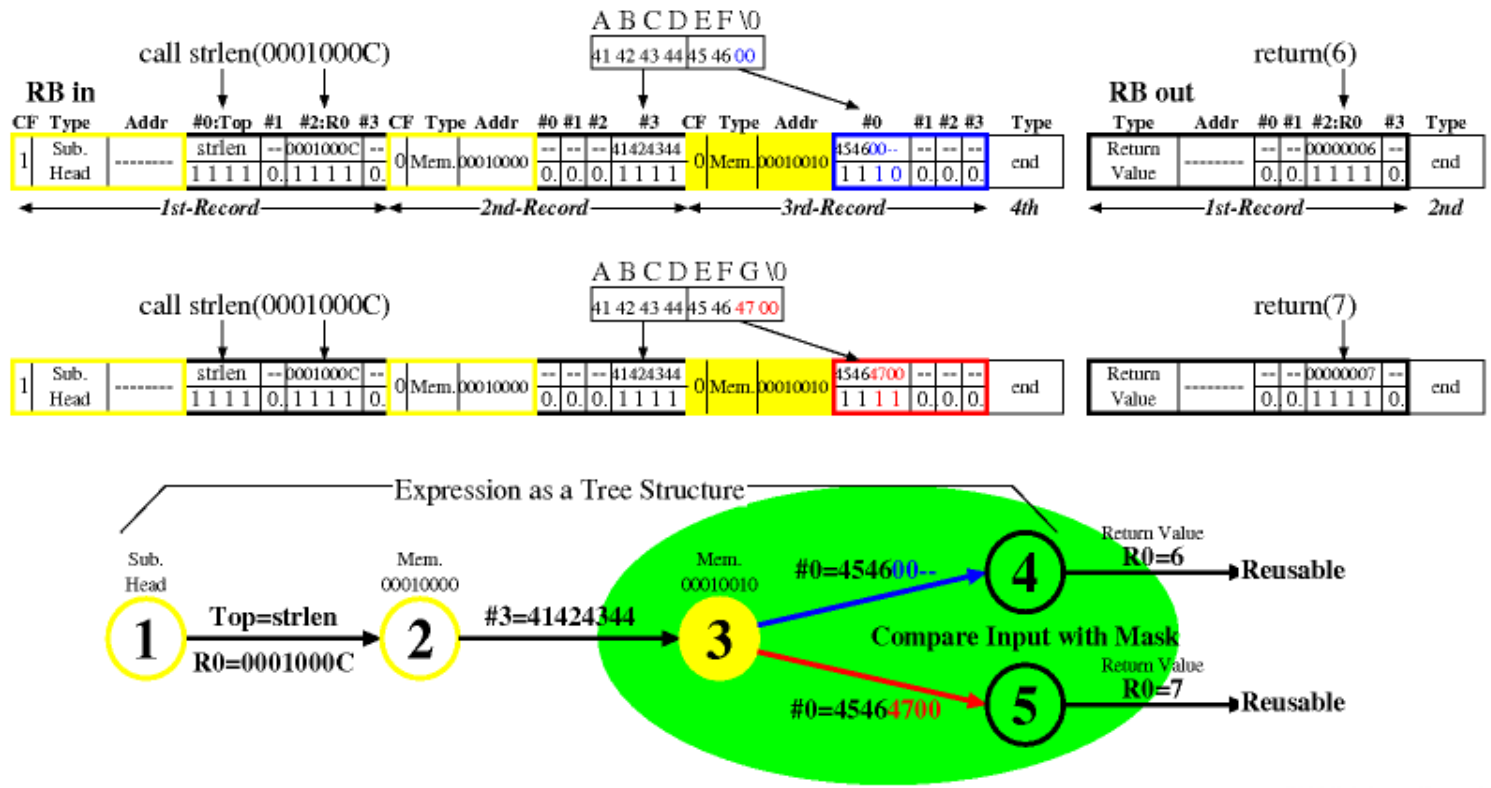
Input ⋯ First-Read-Vals are registered in referenced order
Output ⋯ Final-Write-Vals are registered
strlen("ABCDEF") ⇒ return(6)
strlen("ABCDEFG") ⇒ return(7)

Input can be expressed as a tree.

- Nodes: Register-No./Memory-Address
- Edges: Value

```
strlen(str) char *str;
{
  char *s;
  for (s = str; *s; ++s);
  return (s - str);
}
```

```
strlen:
   move   R0->Rs
   ld.b   [Rs]->Rb
   comp   Rb,0
   beq    end
loop:
   inc    Rs
   ld.b   [Rs]->Rb
   comp   Rb,0
   bneq   loop
end:
   sub    Rs,R0->R0
   return
```

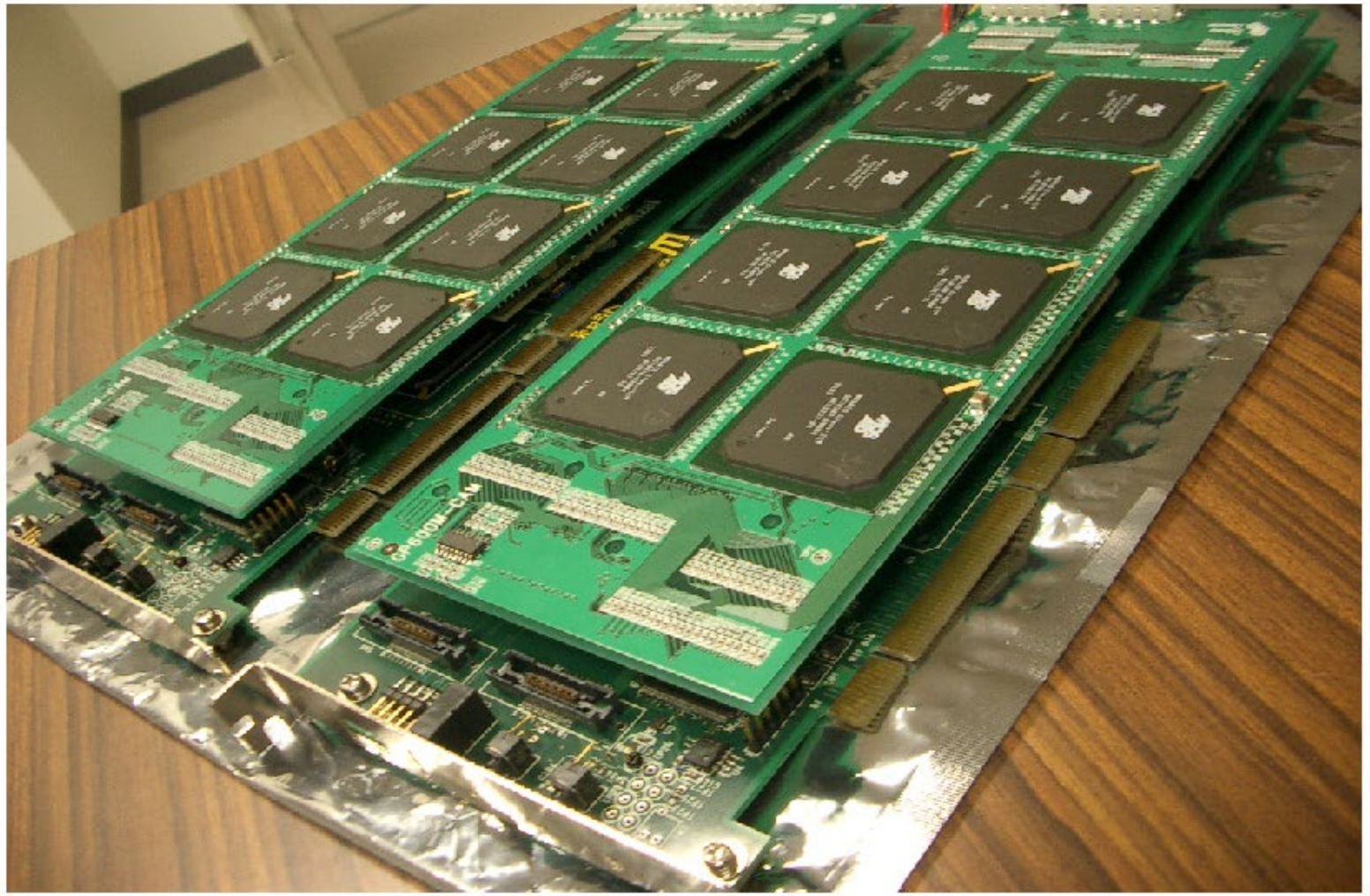


NAIST

2002- 命令区間のin/outをハードウェアがメモし 同じ計算を再利用、さらに未来の計算を予測し投機実行

1 μsec/search, 900 times faster than software

The simulation is accelerated 90 times.



Feasibility of Reuse Memory (RM)

General purpose CAM has separate write/search/read cycle.

Proposed T-CAM has fused operations (Search&Write + Search&Read)

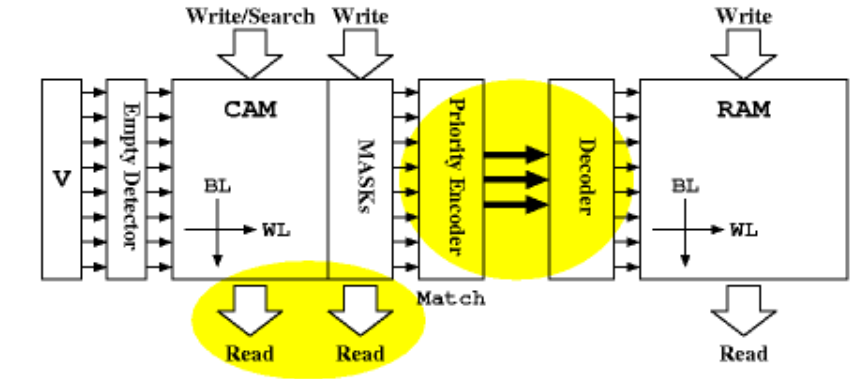
- Search before Write
- Active match-line can be limited to at most one

Search&Read can generate match history for input speculation

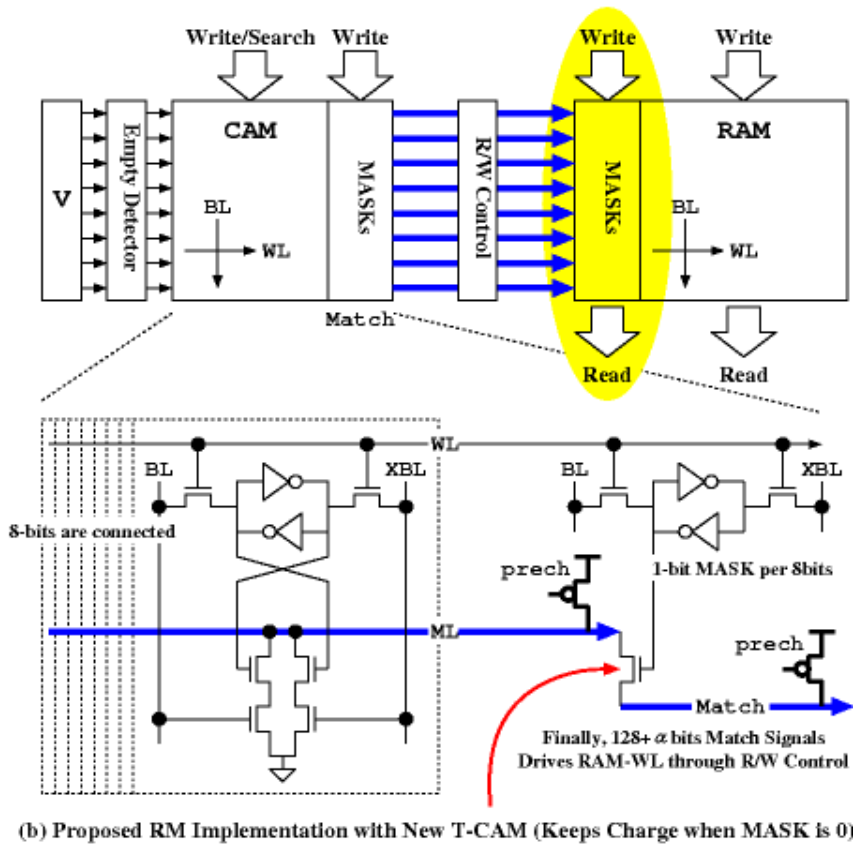
- Fast extraction of CAM by duplicating MASK bits in CAM

Search&Write can warrant single match on CAM

- Fast operation by sharing R/W bit-lines and Comparand bit-lines



(a) RM Implementation with Well-known T-CAM



(b) Proposed RM Implementation with New T-CAM (Keeps Charge when MASK is 0)

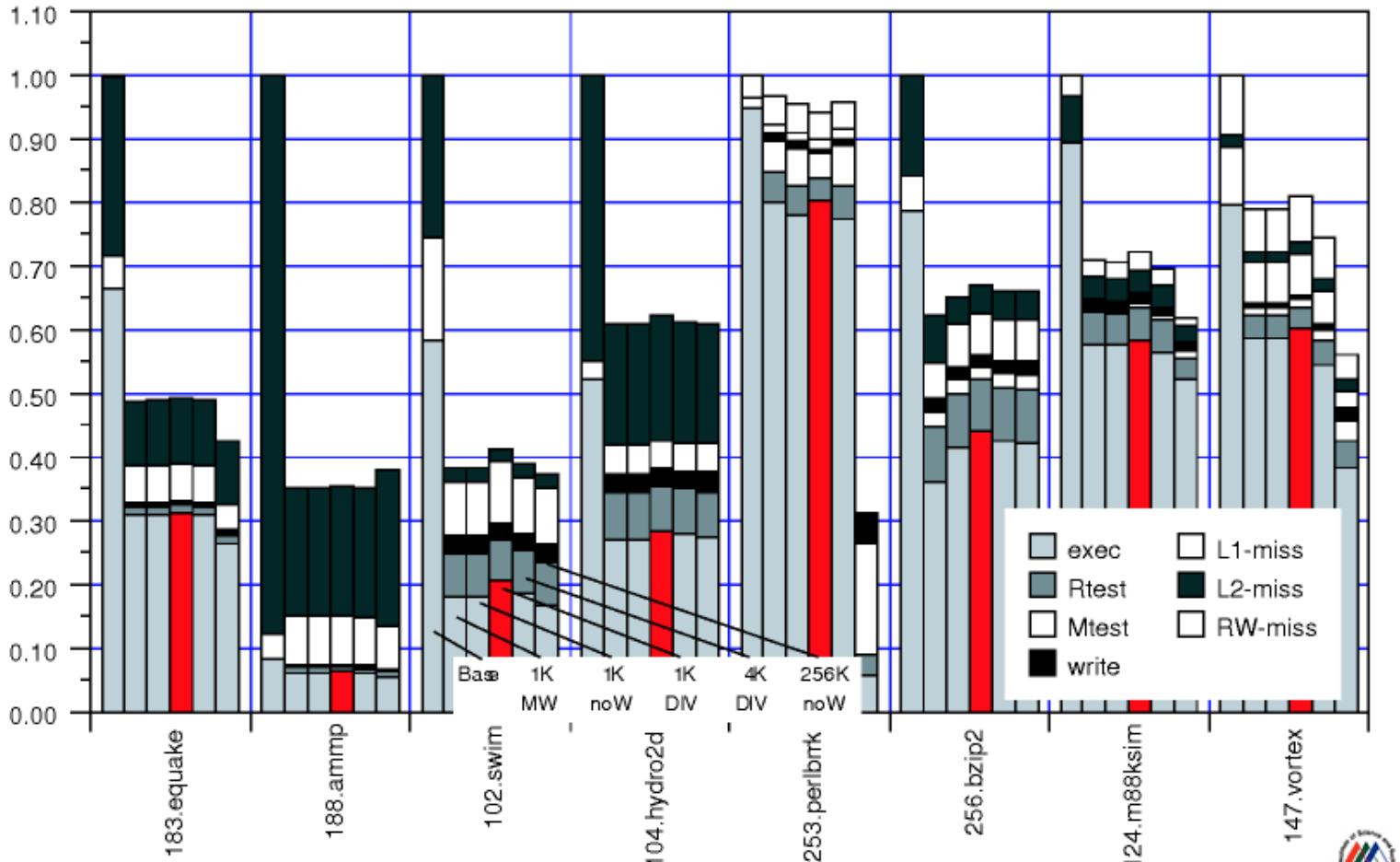
NAIST

回路シミュレーションによる評価

- 現状では，サイクルタイム1. 6nsでの動作を確認(85℃)

Evaluation of Models

Base: 2-way SS SPARC
1K-MW: 16KB Reuse Buffer with SP synchronization
1K-noW: 16KB Reuse Buffer without SP synchronization
1K-DIV: 16KB Reuse Buffer blocked into 64 lines * 16 blocks
4K-DIV: 64KB Reuse Buffer blocked into 256 lines * 16 blocks
256K: 4MB Reuse Buffer no partitioning
Relative cycles based on no-reuse

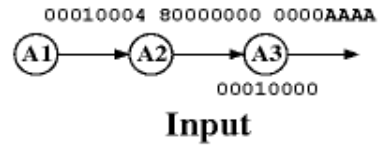


NAIST

投機スレッド間の連携

命令区間間の依存関係

```
loop:(PC=1000)   ループ１回目
1:  set    A1          -> R1
2:  ld     (A1=R1)     -> Rx      ...(00010004)
3:  set    A2          -> R2
4:  ld     (A2=R2)     -> Ry      ...(80000000)
5:  ld     (A3=Rx-4)   -> Rz      ...(0000aaaa)
6:  add    Rx+4        -> Rx      ... 00010008
7:  st     Rx          ->(A1=R1)  ... 00010008
8:  shift  Ry          -> Ry      ... 40000000
9:  st     Ry          ->(A2=R2)  ... 40000000
10: add    Ry+Rz       -> Rz      ... 4000aaaa
11: st     Rz          ->(A4=Rx)  ... 4000aaaa
12: br     loop
```

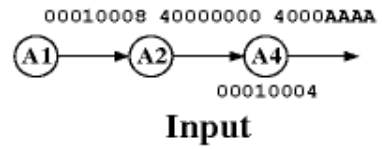


Input

Rx	00010008
Ry	40000000
Rz	4000AAAA
A1	00010008
A2	40000000
A4 (00010004)	4000AAAA

Output

```
loop:(PC=1000)   ループ２回目
set    A1          -> R1
ld     (A1=R1)     -> Rx      ...(00010008)
set    A2          -> R2
ld     (A2=R2)     -> Ry      ...(40000000)
ld     (A4=Rx-4)   -> Rz      ...(4000aaaa)
add    Rx+4        -> Rx      ... 0001000c
st     Rx          ->(A1=R1)  ... 0001000c
shift  Ry          -> Ry      ... 20000000
st     Ry          ->(A2=R2)  ... 20000000
add    Ry+Rz       -> Rz      ... 6000aaaa
st     Rz          ->(A5=Rx)  ... 6000aaaa
br     loop
```

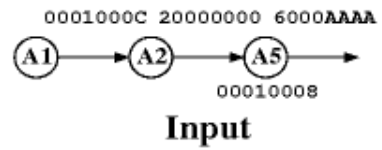


Input

Rx	0001000C
Ry	20000000
Rz	6000AAAA
A1	0001000C
A2	20000000
A5 (00010008)	6000AAAA

Output

```
loop:(PC=1000)   ループ３回目
set    A1          -> R1
ld     (A1=R1)     -> Rx      ...(0001000c)
set    A2          -> R2
ld     (A2=R2)     -> Ry      ...(20000000)
ld     (A5=Rx-4)   -> Rz      ...(6000aaaa)
add    Rx+4        -> Rx      ... 00010010
st     Rx          ->(A1=R1)  ... 00010010
shift  Ry          -> Ry      ... 10000000
st     Ry          ->(A2=R2)  ... 10000000
add    Ry+Rz       -> Rz      ... 7000aaaa
st     Rz          ->(A6=Rx)  ... 7000aaaa
br     loop
```



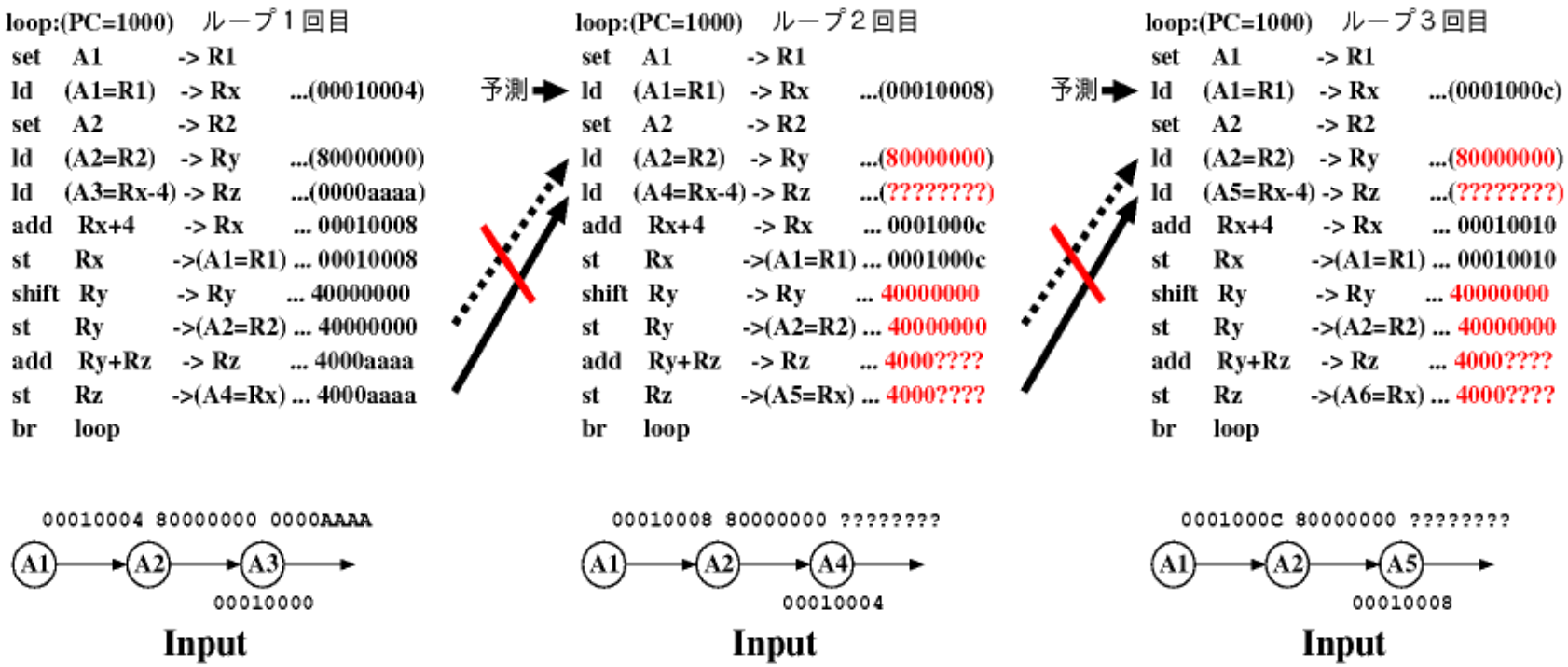
Input

Rx	00010010
Ry	10000000
Rz	7000AAAA
A1	00010010
A2	10000000
A6 (0001000C)	7000AAAA

Output

投機スレッド間連携

さらに進めると，スレッド間連携が可能（以下は，連携しない例）



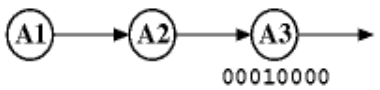
投機スレッド間連携

以下は，連携する例

loop:(PC=1000)　ループ１回目

```
set   A1          -> R1
ld    (A1=R1)     -> Rx      ...(00010004)
set   A2          -> R2
ld    (A2=R2)     -> Ry      ...(80000000)
ld    (A3=Rx-4)   -> Rz      ...(0000aaaa)
add   Rx+4        -> Rx      ... 00010008
st    Rx          ->(A1=R1)  ... 00010008
shift Ry          -> Ry      ... 40000000
st    Ry          ->(A2=R2)  ... 40000000
add   Ry+Rz       -> Rz      ... 4000aaaa
st    Rz          ->(A4=Rx)  ... 4000aaaa
br    loop
```

00010004 80000000 0000**AAAA**



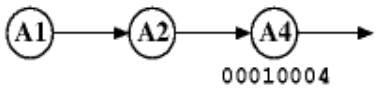
Input

loop:(PC=1000)　ループ２回目

予測

```
set   A1          -> R1
ld    (A1=R1)     -> Rx      ...(00010008)
set   A2          -> R2
                  :
                  :
                  :
                  :
                  :
                  :
                  :
ld    (A2=R2)     -> Ry      ...(40000000)
ld    (A4=Rx-4)   -> Rz      ...(4000aaaa)
add   Rx+4        -> Rx      ... 0001000c
st    Rx          ->(A1=R1)  ... 0001000c
shift Ry          -> Ry      ... 20000000
st    Ry          ->(A2=R2)  ... 20000000
add   Ry+Rz       -> Rz      ... 6000aaaa
st    Rz          ->(A5=Rx)  ... 6000aaaa
br    loop
```

00010008 40000000 4000**AAAA**



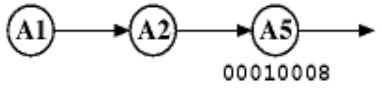
Input

loop:(PC=1000)　ループ３回目

予測

```
set   A1          -> R1
ld    (A1=R1)     -> Rx      ...(0001000c)
set   A2          -> R2
                  :
                  :
                  :
                  :
                  :
                  :
                  :
                  :
                  :
                  :
                  :
                  :
                  :
ld    (A2=R2)     -> Ry      ...(20000000)
ld    (A5=Rx-4)   -> Rz      ...(6000aaaa)
add   Rx+4        -> Rx      ... 00010010
st    Rx          ->(A1=R1)  ... 00010010
shift Ry          -> Ry      ... 10000000
st    Ry          ->(A2=R2)  ... 10000000
add   Ry+Rz       -> Rz      ... 7000aaaa
st    Rz          ->(A6=Rx)  ... 7000aaaa
br    loop
```

0001000C 20000000 6000**AAAA**



Input

NAIST

連携のための動的アドレス解析

- 定数アドレスであり内容が変化しない.
  再利用テストの観点からはそもそも処理対象外である.

- 定数アドレスであり内容の変化量が一定である.
  予測が可能である．前述のプログラム例におけるアドレスＡ１が該当する.

- 定数アドレスであり内容の変化量が不定である.
  予測は困難なので，書き込みを待ち合わせる必要がある．前述のプログラム例におけるアドレスＡ２が該当する.

- 変化するアドレスであるものの，個々のアドレスについてはストアは発生せず内容が変化しない.
  再利用テストの観点からはそもそも処理対象外である.

- 変化するアドレスであり，個々のアドレスについてストアが発生.
  変化量が一定であることは期待できず予測は困難なので，書き込みを待ち合わせる必要がある．前述のプログラム例におけるアドレスＡ３～Ａ１０が該当する.

連携のための動的アドレス解析

履歴の保存

ＭＳＰ／ＳＳＰ毎入出力記録

Const-FLAG

R1	1
R2	1
Rx	0
Ry	0
Rz	0

	Register/Memory Read			Register/Memory Write						
	#1	#2	#3		#1	#2	#3	#4	#5	#6
C-FLAG	change	change								
P-Mask	FFFFFFFF	FFFFFFFF	00000000	Address	Rx	Ry	Rz	A1	A2	A4
Address	A1	A2	A3	Mask	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF
Mask	FFFFFFFF	FFFFFFFF	FFFFFFFF	S-Count				0001	0001	0001
Value	00010004	80000000	0000AAAA	Value	00010008	40000000	4000AAAA	00010008	40000000	4000AAAA

ＭＳＰの実行／再利用に基づく実際の命令区間毎History

Address	A1	A2	A3	Address	A1	A2	A4
P-Mask	FFFFFFFF	FFFFFFFF	00000000	Mask	FFFFFFFF	FFFFFFFF	FFFFFFFF
S-Count	0001	0001	0001	S-Count	0001	0001	0001
Value	00010004	80000000					

ループ１回目

Address	A1	A2	A4	Address	A1	A2	A5
P-Mask	FFFFFFFF	FFFFFFFF	00000000	Mask	FFFFFFFF	FFFFFFFF	FFFFFFFF
S-Count	0001	0001	0001	S-Count	0001	0001	0001
Value	00010008	40000000					

ループ２回目

Address	A1	A2	A5	Address	A1	A2	A6
P-Mask	FFFFFFFF	FFFFFFFF	00000000	Mask	FFFFFFFF	FFFFFFFF	FFFFFFFF
S-Count	0001	0001	0001	S-Count	0001	0001	0001
Value	0001000C	20000000					

ループ３回目

Address	A1	A2	A6	Address	A1	A2	A7
P-Mask	FFFFFFFF	FFFFFFFF	00000000	Mask	FFFFFFFF	FFFFFFFF	FFFFFFFF
S-Count	0001	0001	0001	S-Count	0001	0001	0001
Value	00010010	10000000					

ループ４回目

連携のための動的アドレス解析

履歴に基づくアドレスの分類と待ち合わせ

ＭＳＰの実行／再利用に基づく実際の命令区間毎History

Address	A1	A2	A3
P-Mask	FFFFFFFF	FFFFFFFF	00000000
S-Count	0001	0001	0001
Value	00010004	80000000	

ループ１回目

diff=04　diff=-4

Address	A1	A2	A4
P-Mask	FFFFFFFF	FFFFFFFF	00000000
S-Count	0001	0001	0001
Value	00010008	40000000	

ループ２回目

diff=04　diff=-2

Address	A1	A2	A5
P-Mask	FFFFFFFF	FFFFFFFF	00000000
S-Count	0001	0001	0001
Value	0001000C	20000000	

ループ３回目

diff=04　diff=-1

Address	A1	A2	A6
P-Mask	FFFFFFFF	FFFFFFFF	00000000
S-Count	0001	0001	0001
Value	00010010	10000000	

ループ４回目

距離の推定　diff=04　diff=??

ＳＳＰのための予測値　待ち合わせる主記憶値

Address	A1	A2	A7
Mask	FFFFFFFF	FFFFFFFF	FFFFFFFF
S-Count		0000	0001
Value	00010014	WAIT	WAIT

ループ５回目
予測距離＝１　⇒ＭＳＰへ割当

Address	A1	A2	A8
Mask	FFFFFFFF	FFFFFFFF	FFFFFFFF
S-Count		0001	0001
Value	00010018	WAIT	WAIT

ループ６回目
予測距離＝２　⇒ＳＳＰ＃１へ割当

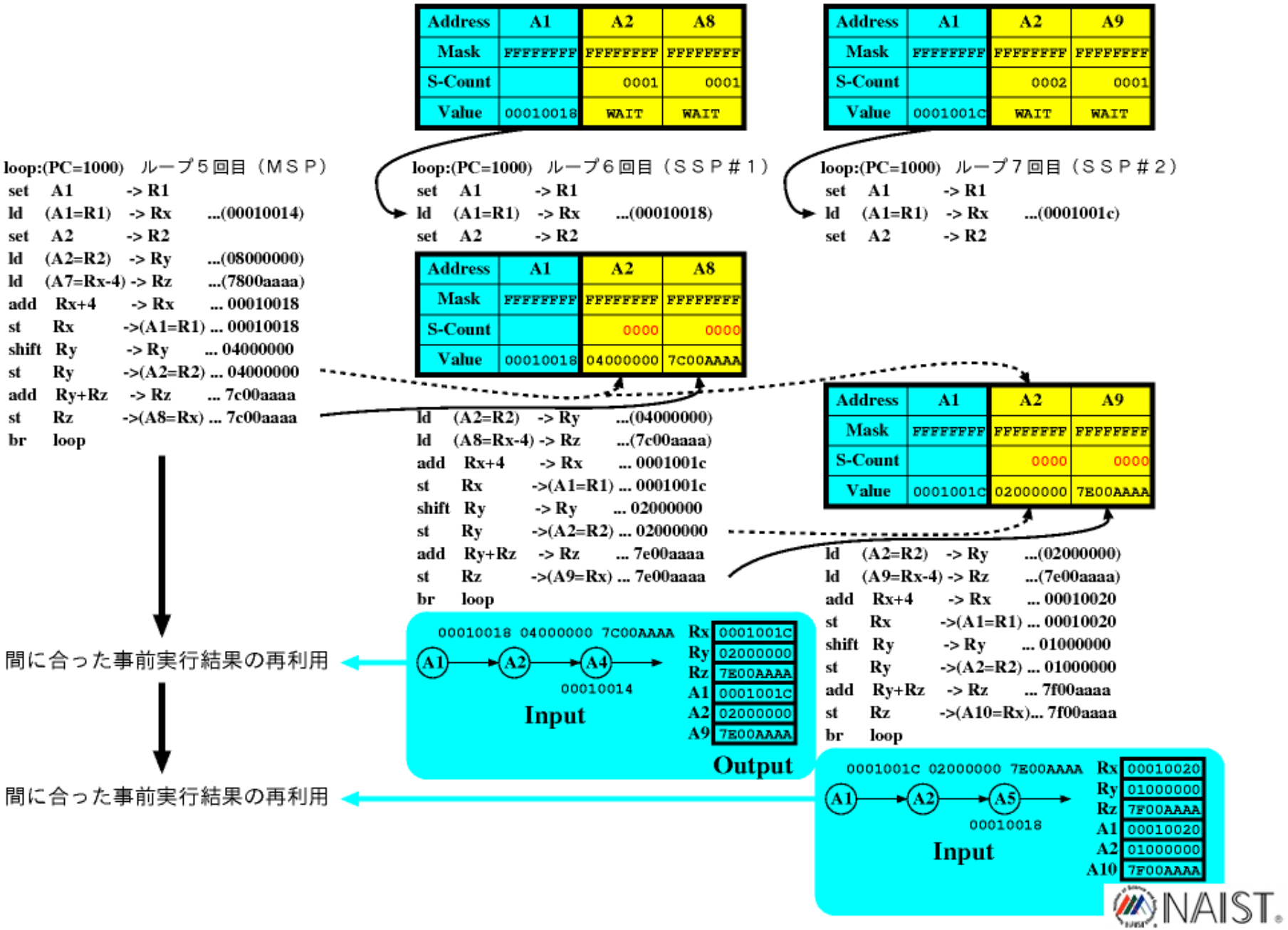
Address	A1	A2	A9
Mask	FFFFFFFF	FFFFFFFF	FFFFFFFF
S-Count		0002	0001
Value	0001001C	WAIT	WAIT

ループ７回目
予測距離＝３　⇒ＳＳＰ＃２へ割当

Address	A1	A2	A10
Mask	FFFFFFFF	FFFFFFFF	FFFFFFFF
S-Count		0003	0001
Value	00010020	WAIT	WAIT

ループ８回目
予測距離＝４　⇒ＳＳＰ＃３へ割当

アドレス解析に基づく連携



NAIST

今日のまとめ

投機実行を大きく分けると，
- 命令の先読みに関するもの
- オペランドアドレスの先読みに関するもの
- オペランド値の先読みに関するもの

必ず儲かることを期待して，様々な「投機」が提案されている
- 一般に，理想値が大きな機構ほど，ハードウェアが複雑化し，失敗時のペナルティも大きいと考えるべき．

ハードウェアによる投機には限界があり，コンパイラによる投機に期待がかかる
- ただし，プログラムが異常終了する可能性がない限り，コンパイラは安全なコードを生成するしかなく，意図した性能は出ない．

今日はここまで