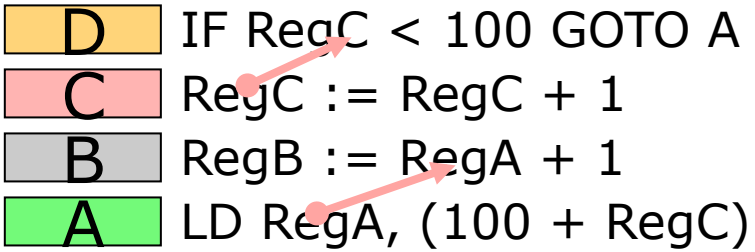
HW Optimizations

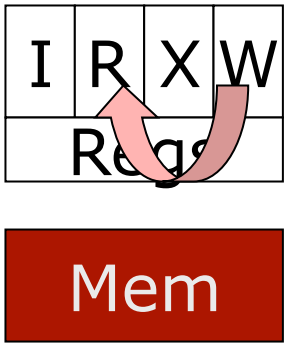
Erik Hagersten
Uppsala University

Outline of these lectures

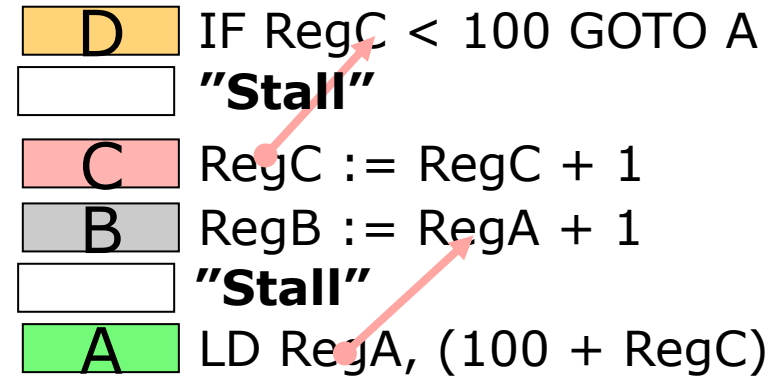
1. Processor implementations
2. Caches and memory system
3. Multiprocessors
4. **HW optimizations**
5. Multicore processors
6. SW optimizations

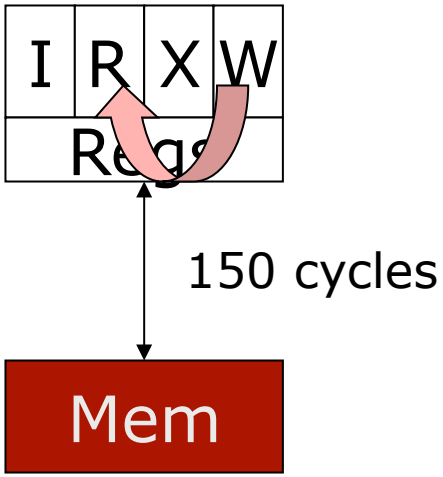
Problem: Data dependence ☹



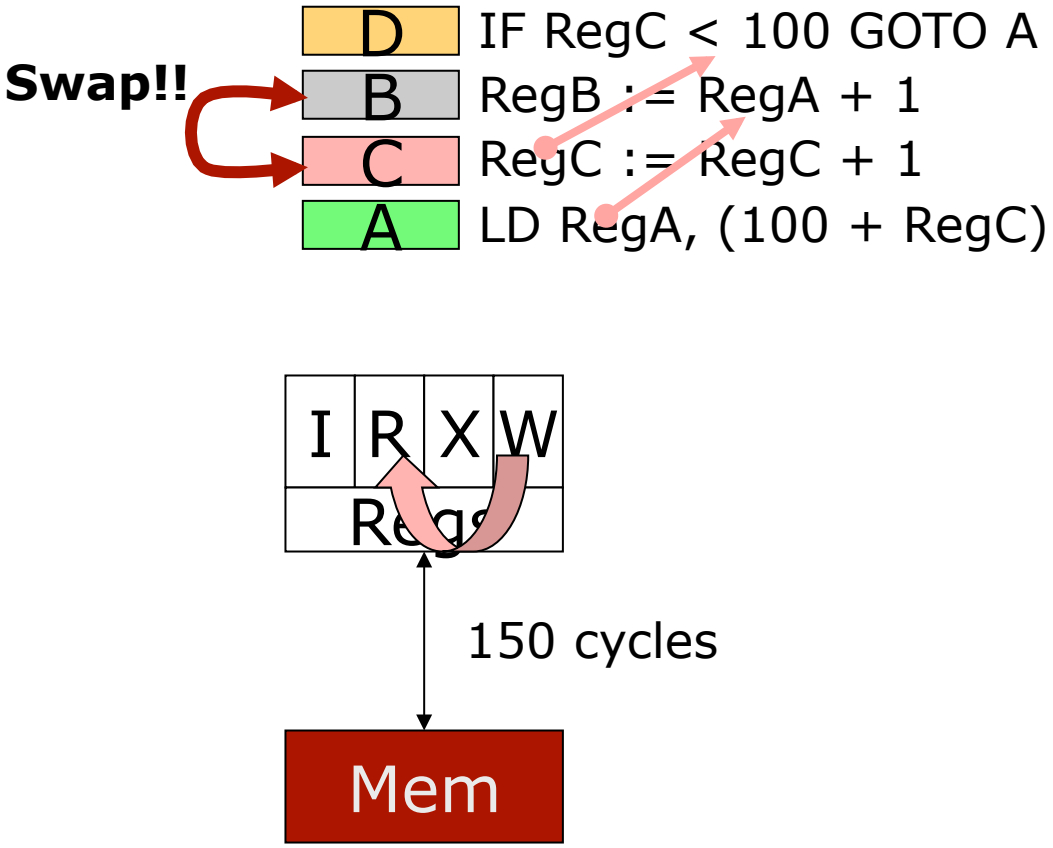


Data dependence fix 1: pipeline delays

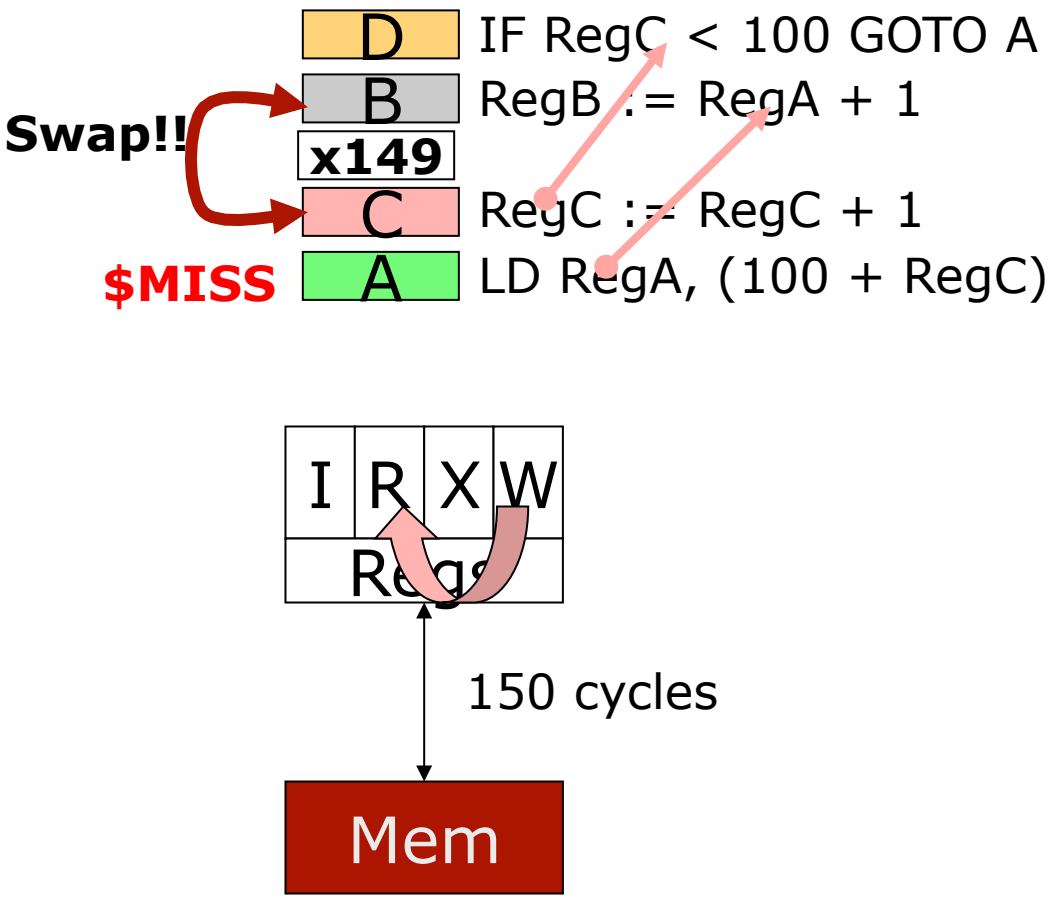




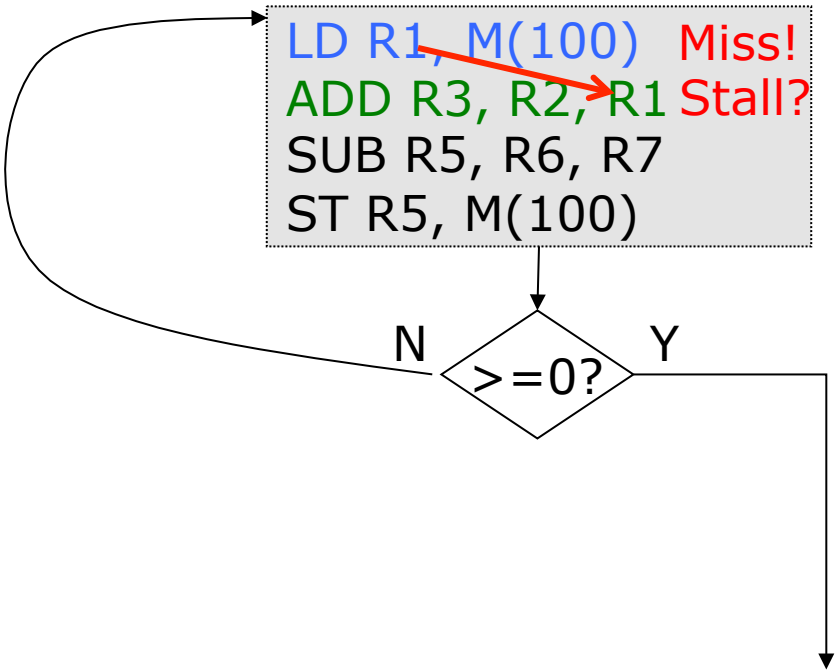
Fix2: Compiler optimizations



Fix2 does not help long delays



Fix3: Out-of-order execution Improving ILP



Out-of-order (OoO) execution:
Execute instructions in ANY order
Make "side-effects" visible in-order
- Update registers
- Change value in memory

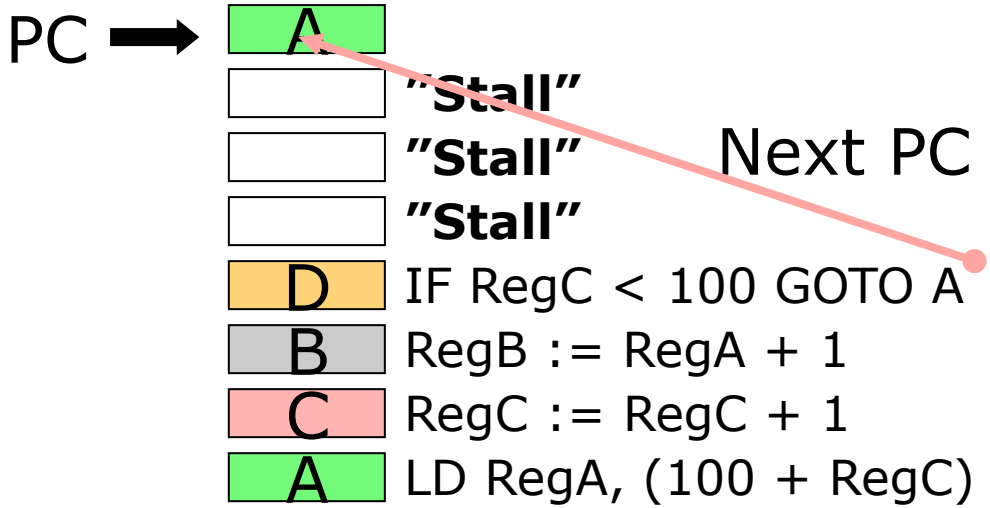
(- Handle exceptions…)

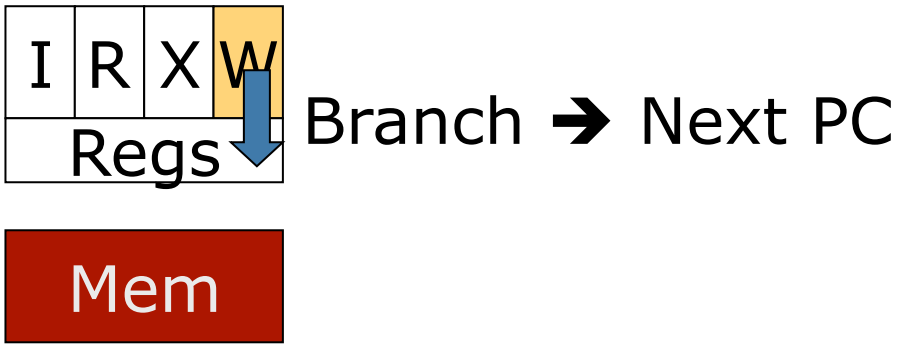
→ **Looks like an in-order execution**

Example:
Assume that LD takes a long time.
The ADD is dependent on the LD ☹
Start the SUB and ST before the ADD
Update R5 and M(100) **after** R3

➔ **Can remove pipeline bubbles**

Problem: Branch delays ☹

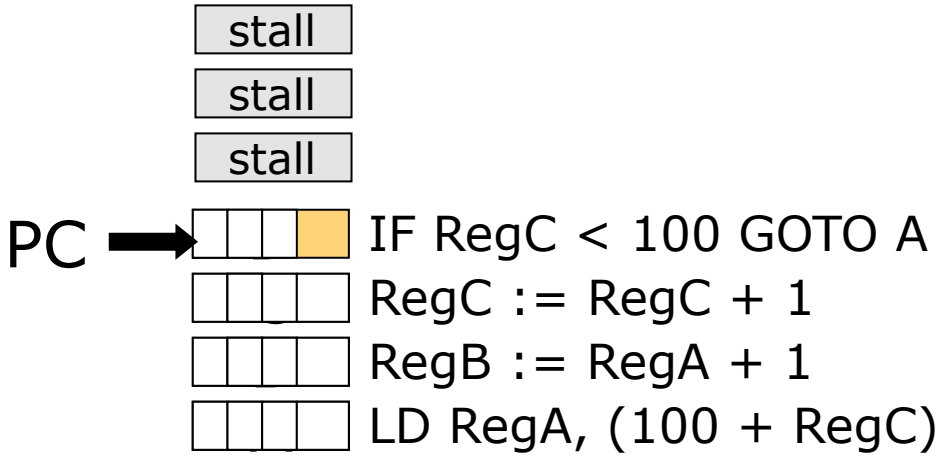


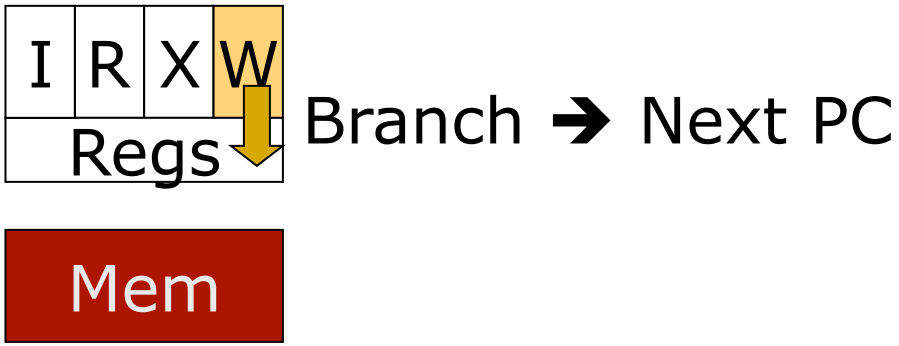


7 cycles per iteration of 4 instructions ☹

Need longer basic blocks with independent instr.

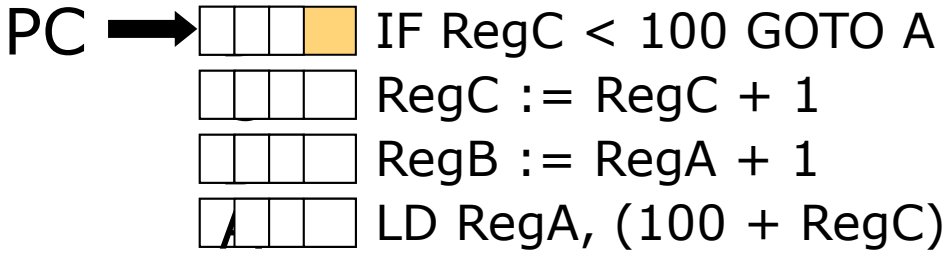
Can we do the branch earlier?



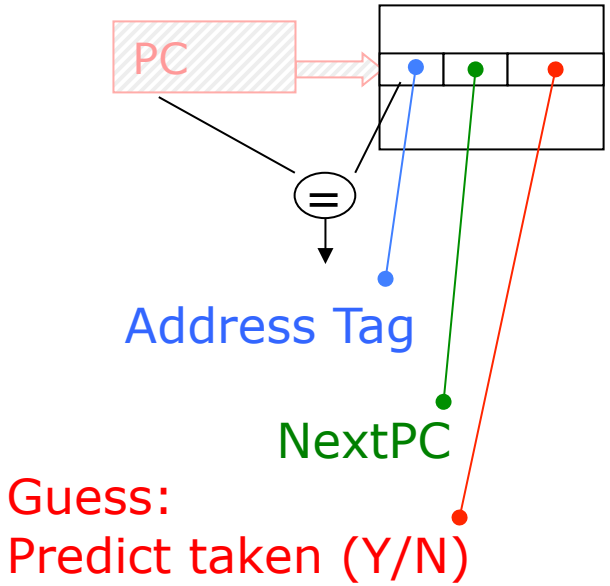


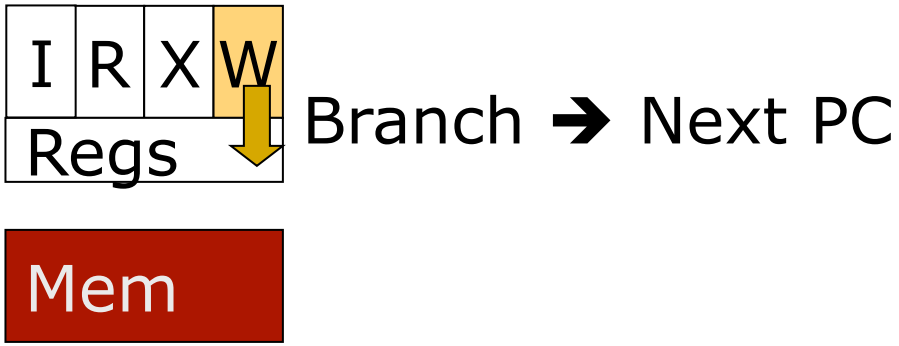
Fix4: Branch Predictor Based on History

Guess the next PC here!!

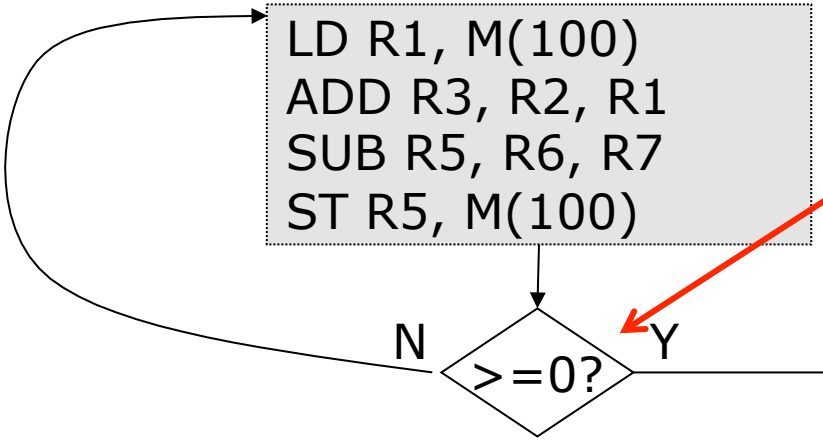


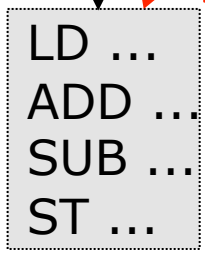
BranchTarget Buffer
(a cache with branch history)





Fix4: Branch prediction



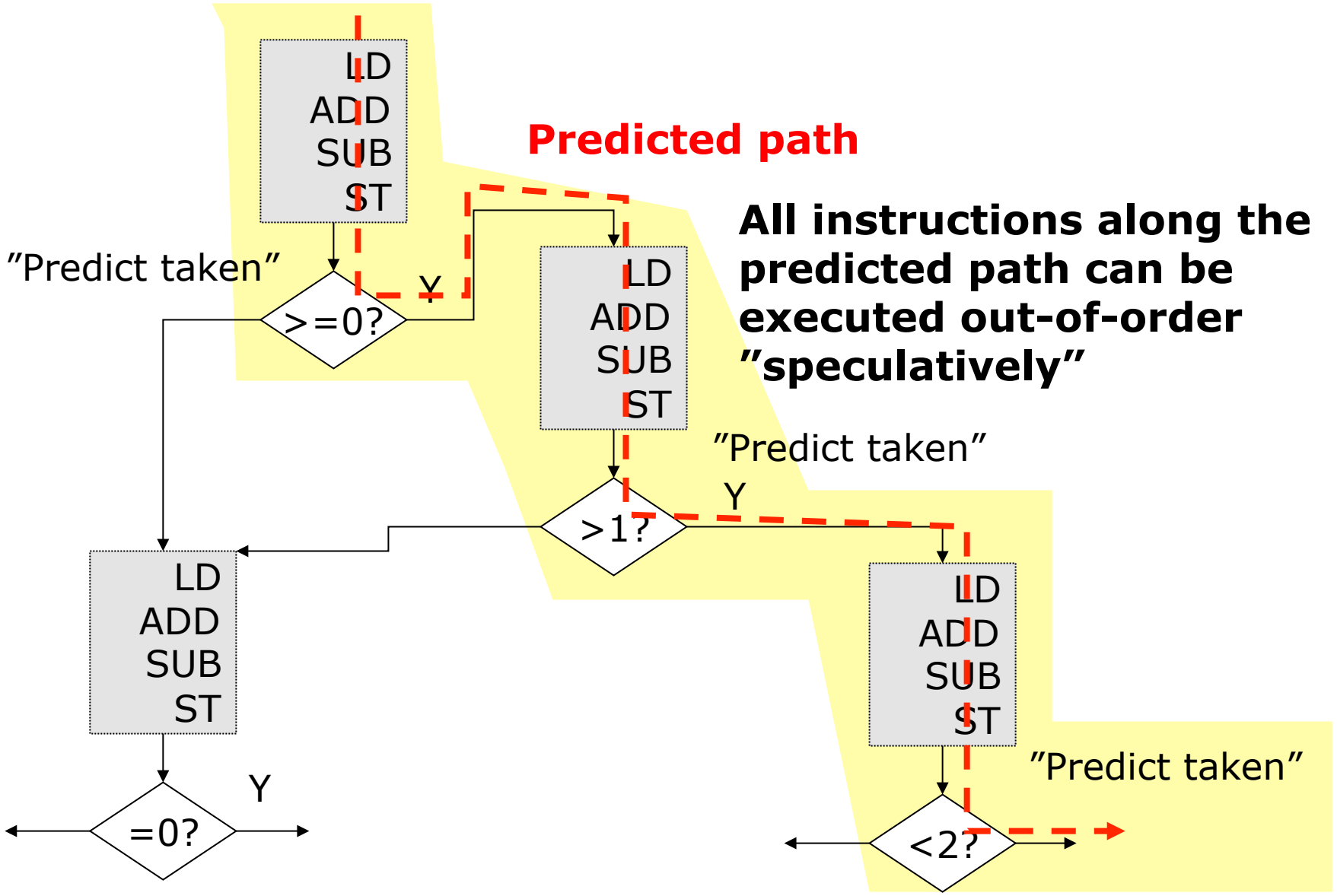


The HW can guess if the branch is taken or not and avoid branch stalls if the guess is correct.

Assume the guess is ”Y”.

The HW can start executing these instruction **before** the outcome of the branch is known, but cannot allow any ”side-effect” to take place until the outcome is known.

Fix5: OoO execution past branches



All instructions along the predicted path can be executed out-of-order "speculatively"

Fix5: Scheduling Past Branches Undo speculative work

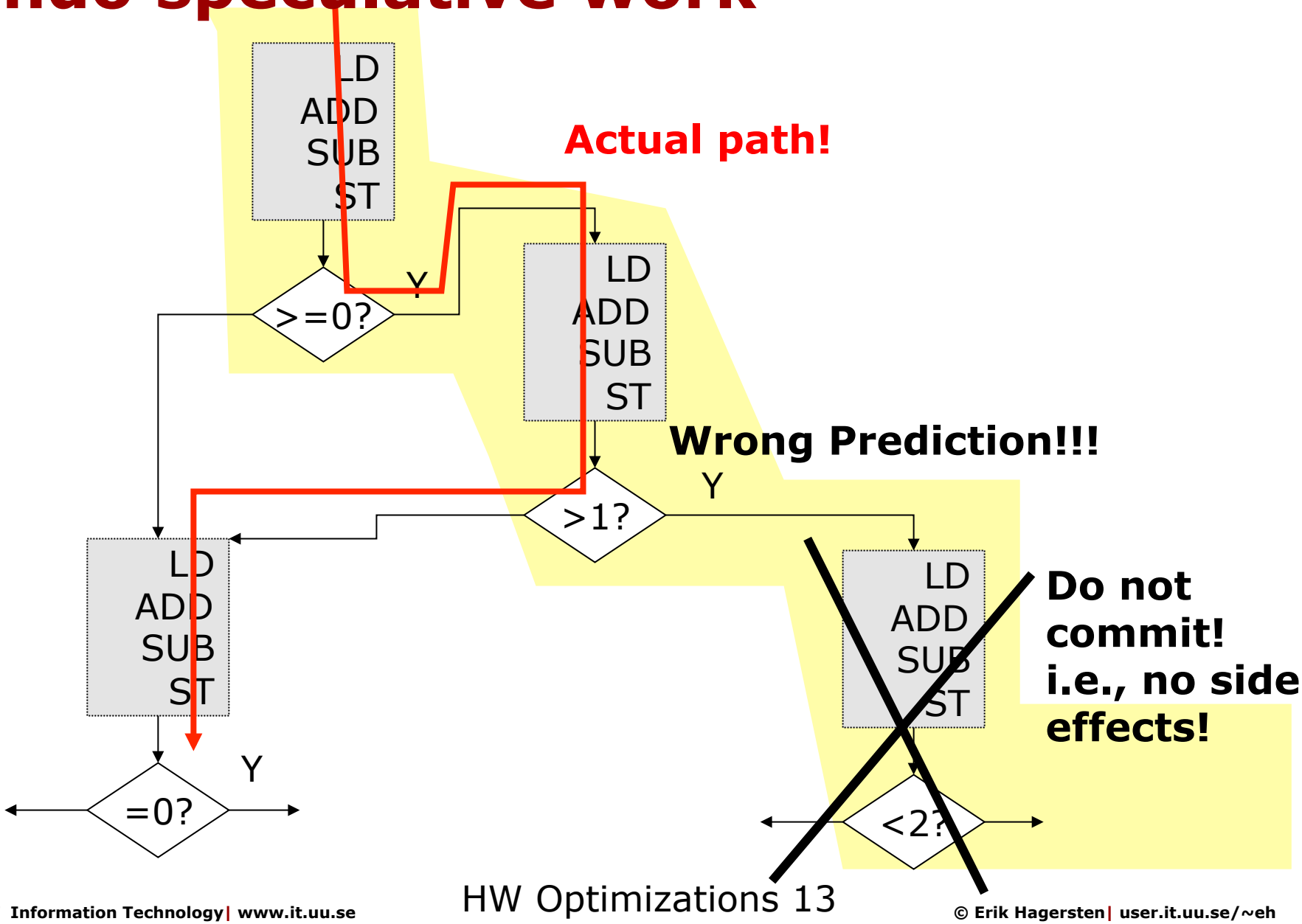
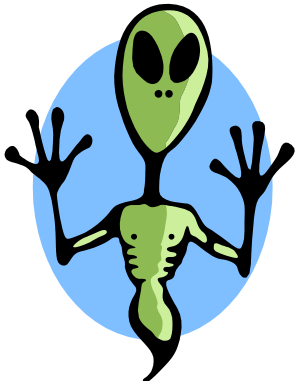


Fig 6: HW prefetching



...a little green man that anticipates your next memory access and prefetches the data to the cache.

Improves Memory-level parallelism (MLP)!

- **Sequential prefetching:** Sequential streams [to a page]. Some number of prefetch streams supported. Often only for L2 and L3.
- **PC-based prefetching:** Detects strides from the same PC. Most often for L1.
- **Adjacent prefetching:** On a miss, also bring in the "neighboring" cache line. Often only for L2 and L3.

Fix 7: SW prefetching

- Special instruction allow the programmer to specify what to prefect
- Typically not supported in HL languages

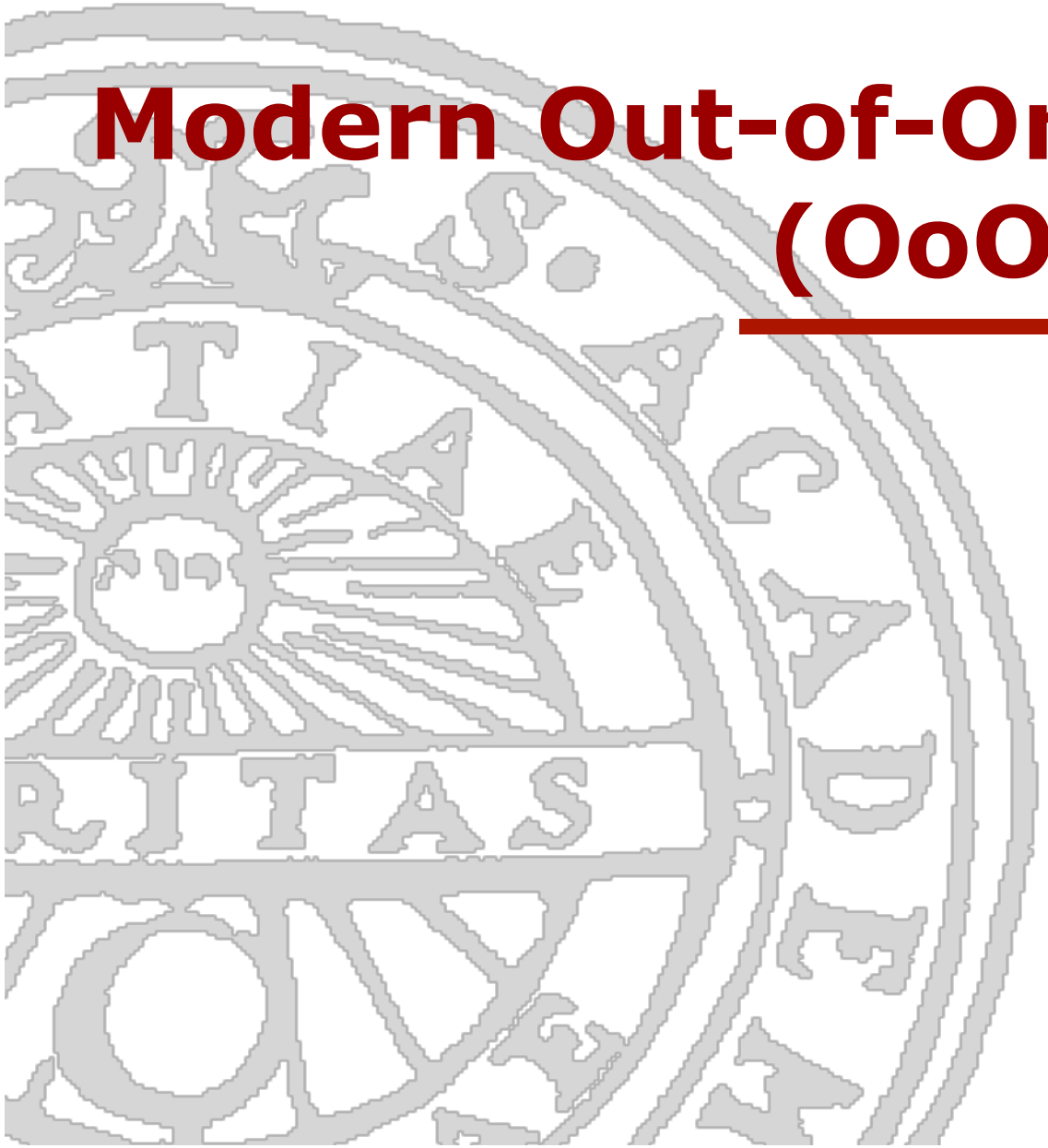
- **No prefetching**

```
for (i = 0; i < N; i++) {
   sum += a[i]*b[i];
}
```

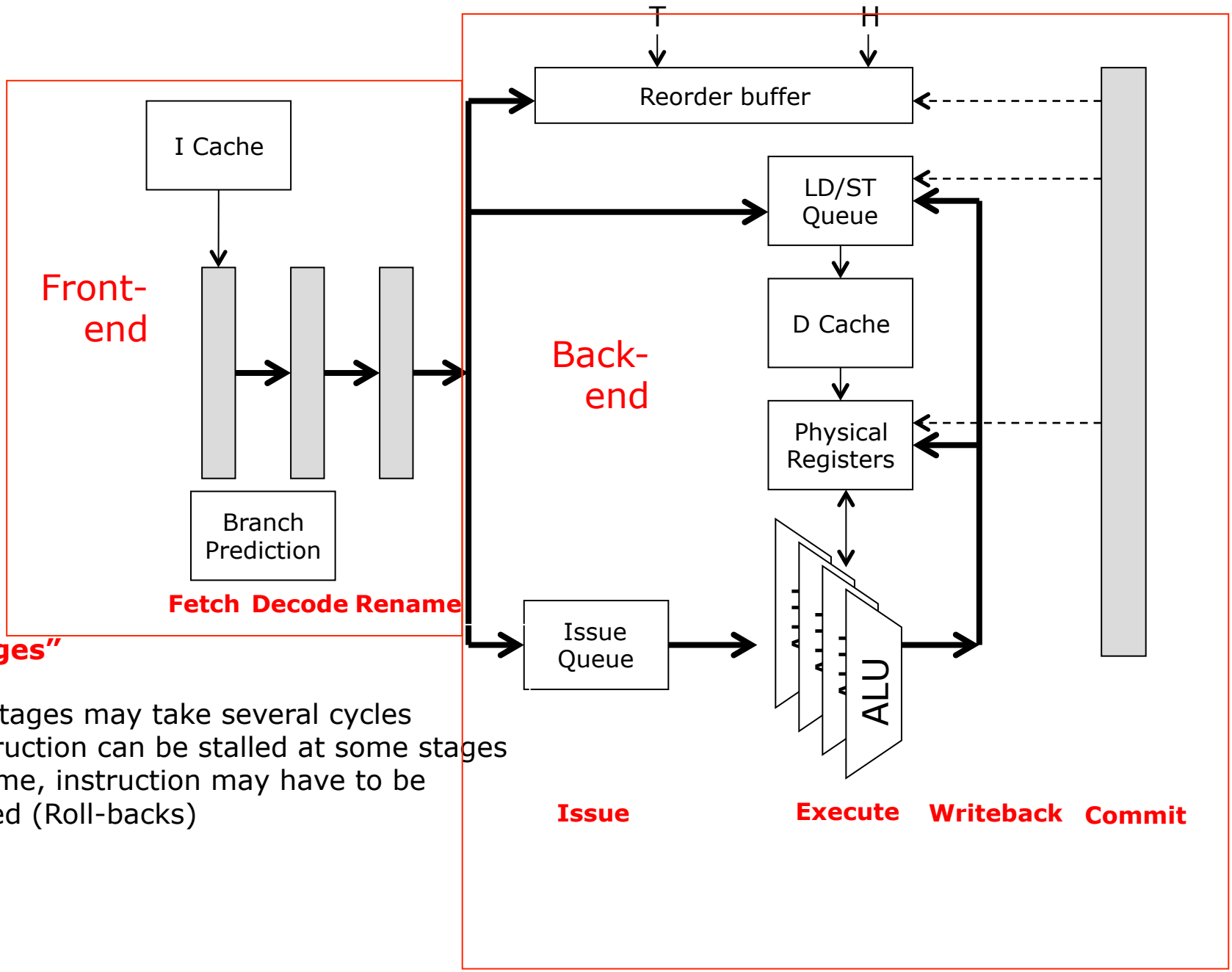
- **Simple pref.**

```
for (i = 0; i < N; i+
  +) {
  fetch (&a[i+PD]);
  fetch (&b[i+PD]);
  sum += a[i]*b[i];
}
```

Prefetch distance

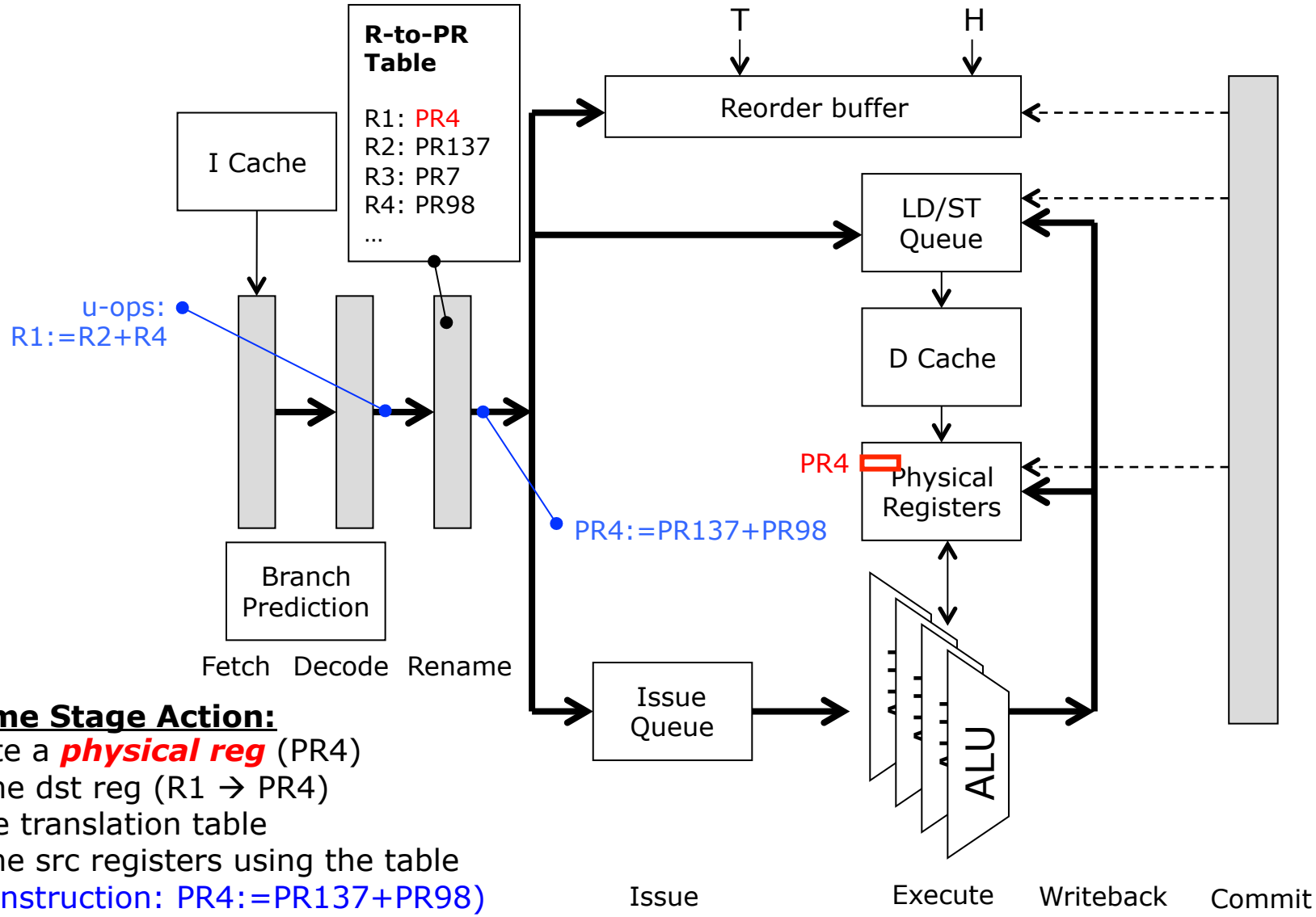


Modern Out-of-Order Pipelines (OoO)



7 "Stages"

Some stages may take several cycles
An instruction can be stalled at some stages
Sometime, instruction may have to be restarted (Roll-backs)



Rename Stage Action:

Allocate a ***physical reg*** (PR4)

Rename dst reg (R1 → PR4)

Update translation table

Rename src registers using the table

(New instruction: PR4:=PR137+PR98)

Flow of Instructions Through Rename

(Both FP and INT Architectural register starting with an R)

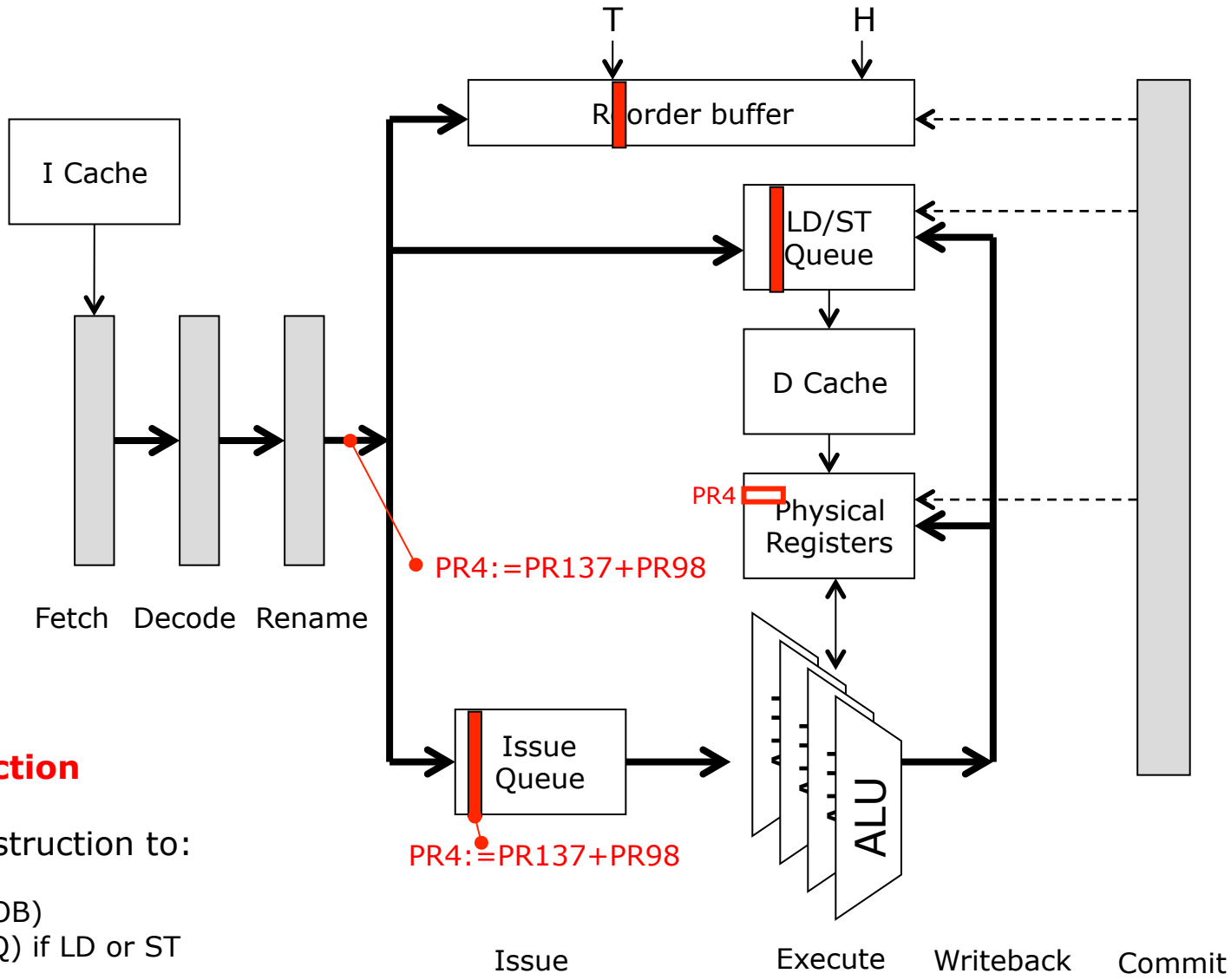
```
1:      LDD   P1 , 0 (Px)     // R4 = a[i]  L1 hit, D on pI3
2:      ADDD  P2 , Py , P1    // sum +=     Depends on I1, pI2
3:      SUBI  P3 , Px , #8    // i++        Depends on pI3
4:      BEQZ  P3 , #LOOP      // last time? Depends on I3
1:      LDD   P4 , 0 (P3)     // R4 = a[i]  L1 hit, D on pI3
2:      ADDD  P5 , P2 , P4    // sum +=     Depends on I1, pI2
3:      SUBI  P6 , P3 , #8    // i++        Depends on pI3
4:      BEQZ  P6 , #LOOP      // last time? Depends on I3
1:      LDD   P7 , 0 (P6)     // R4 = a[i]  L1 hit, D on pI3
....
```

2nd iteration

Rename Table: Mapping from Architectural and Physical Registers
(initially R1 maps to Px and R4 maps to Py)

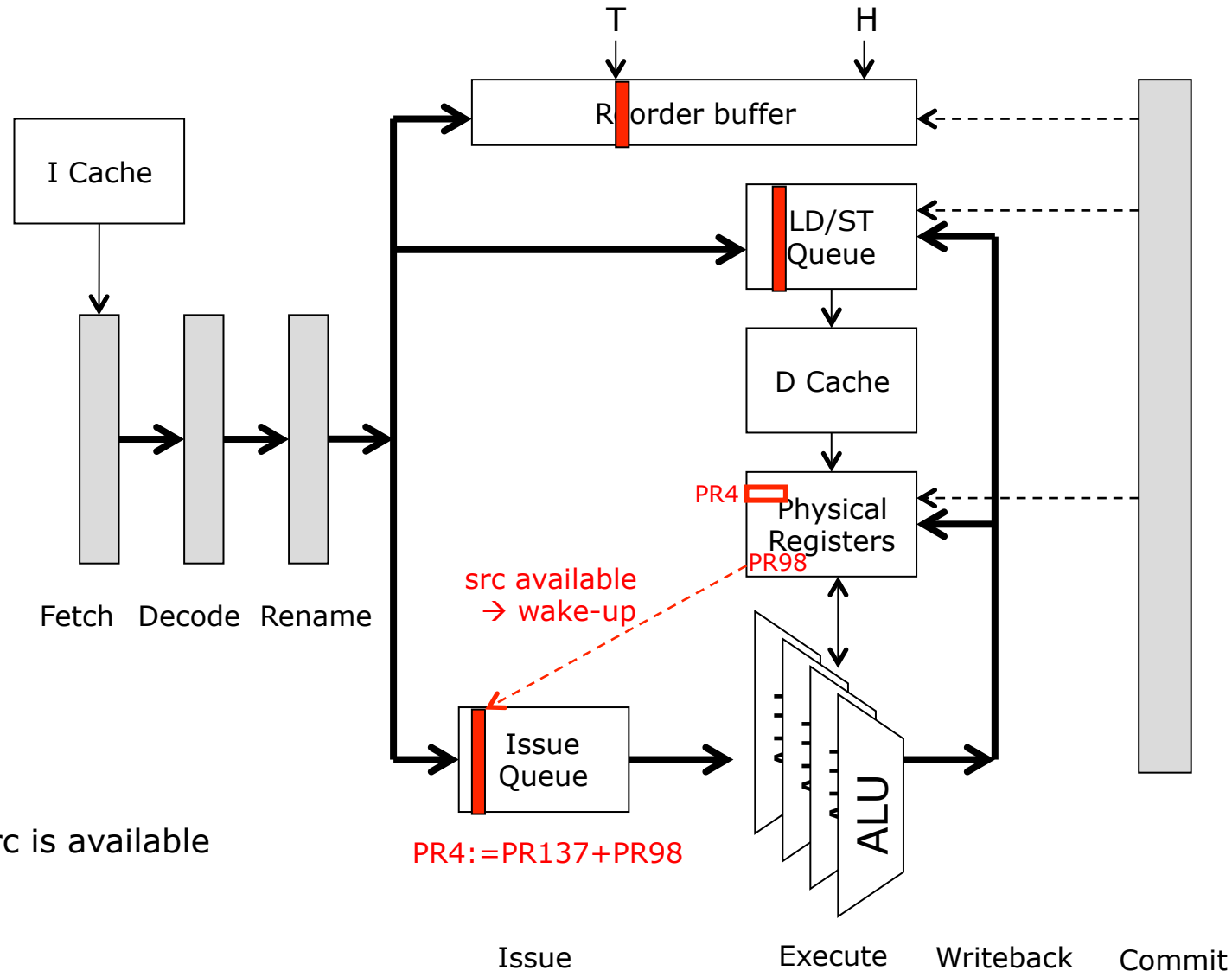
R to PR	
R1:	P6
R2:	
R3:	
R4	P7
R5:	
R6:	P5
…	

Physical registers are write-once
≈5-10X more physical regs than architecture regs
(There is a reuse mechanism for physical registers)
All "true" data dependence is maintained (RAW)
"Name dependence" (WAW and WAR) removed



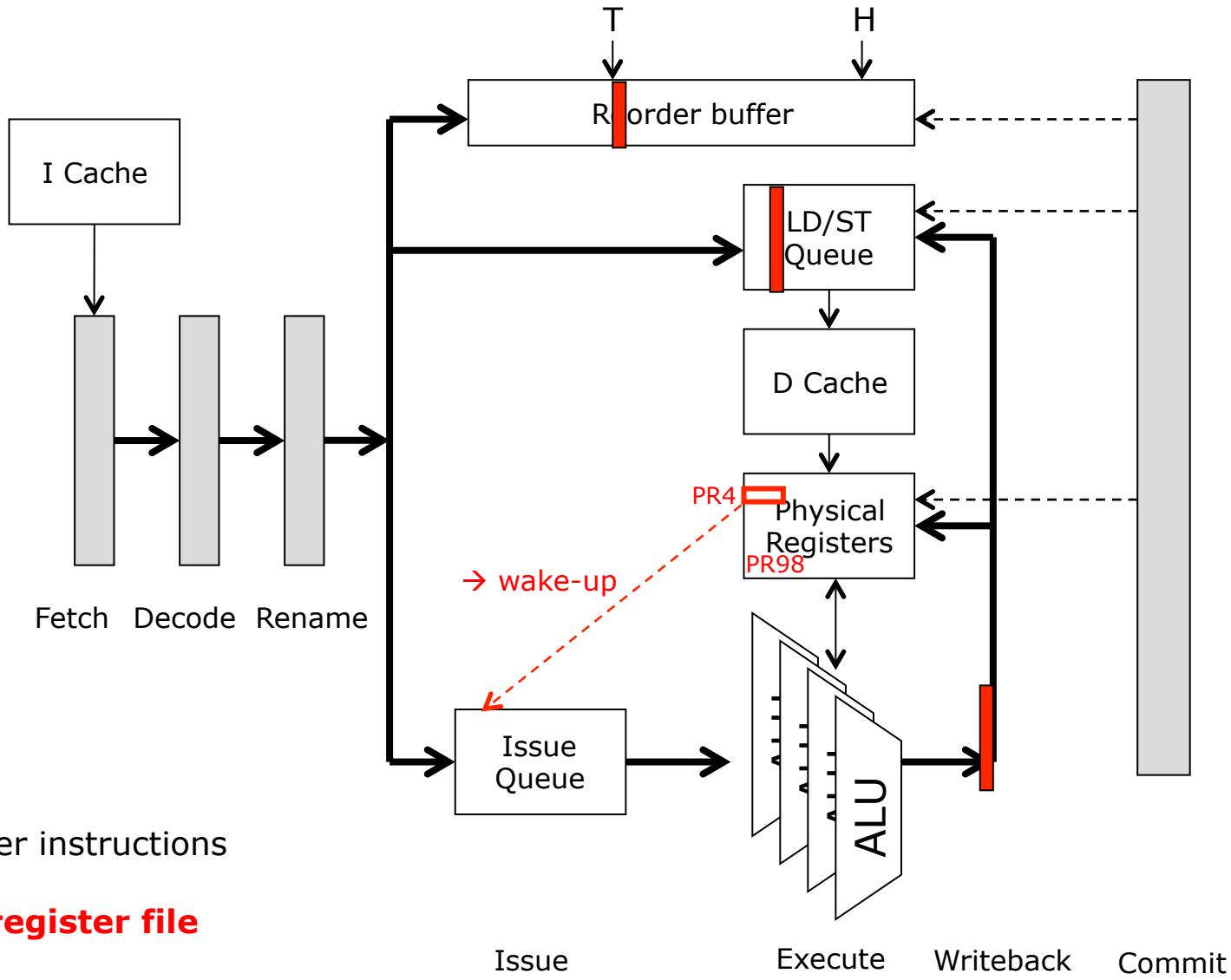
Dispatch instruction

Send renamed instruction to:
- -Issue Queue (IQ)
- -Reorder buffer (ROB)
- -LD/ST Queue (LSQ) if LD or ST



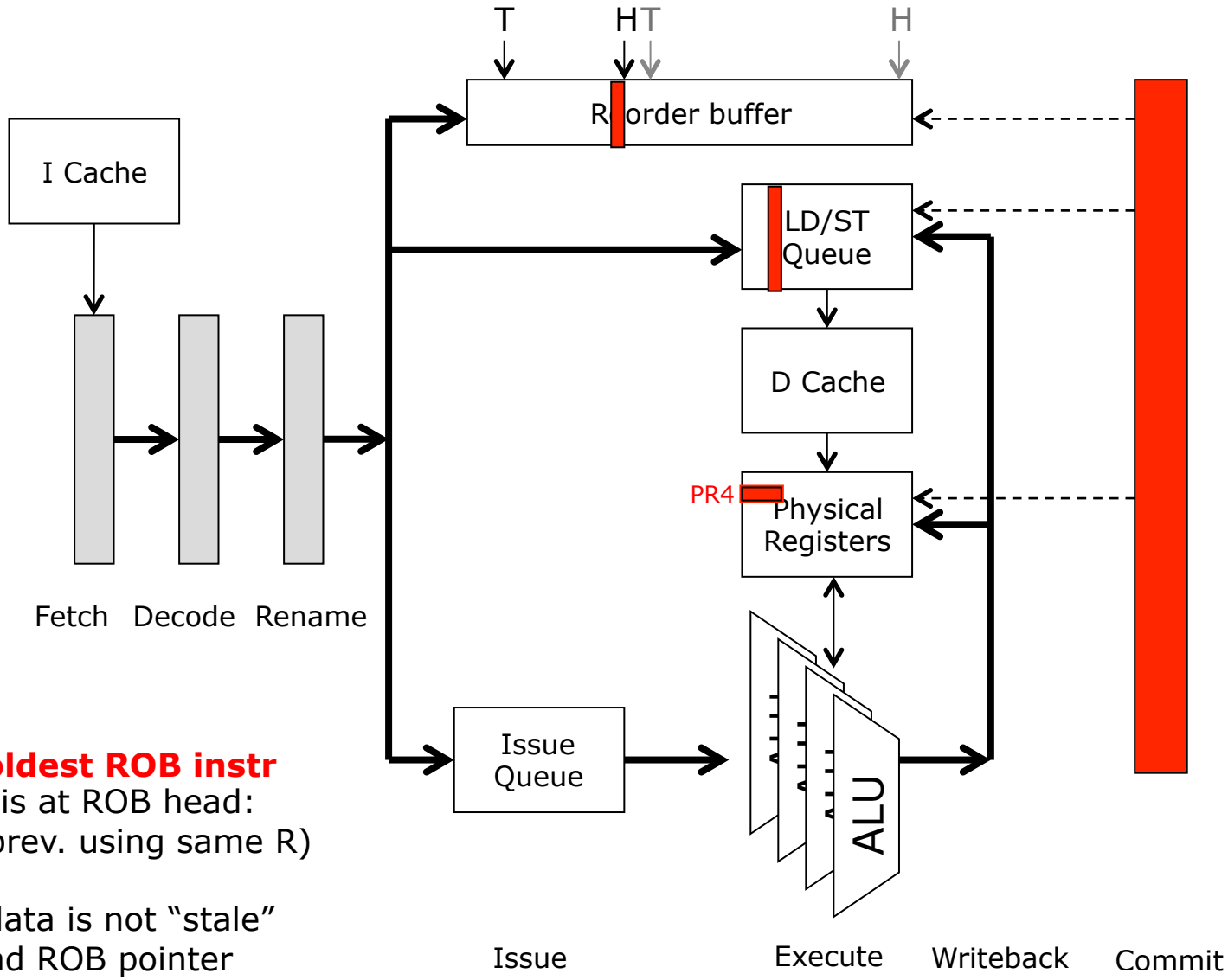
Wake-up in IQ

Wake-up when src is available



Writeback
Update dst PR
May wake-up other instructions

Large physical register file



Commit of the oldest ROB instr

When instruction is at ROB head:

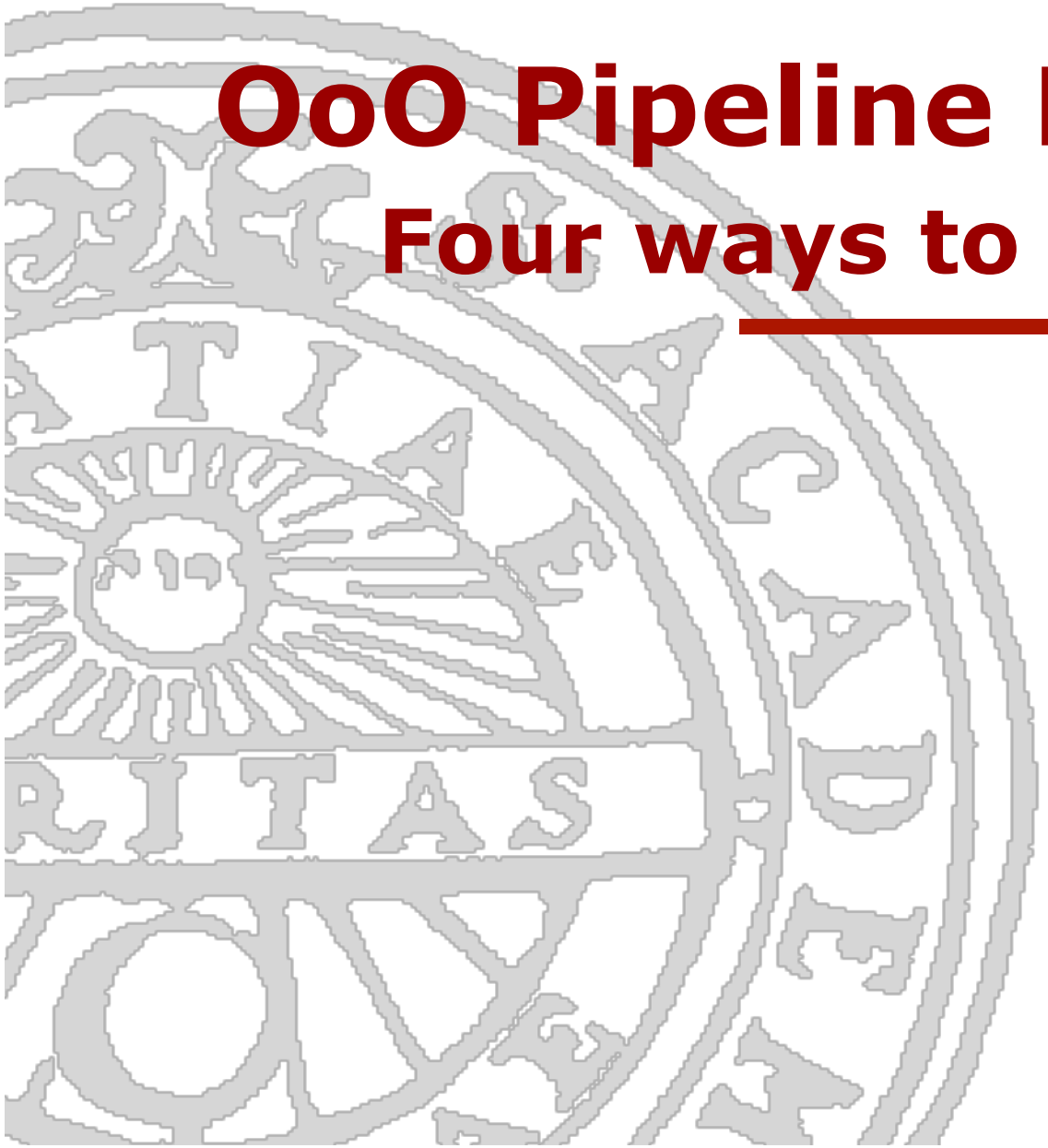
- -Free a PR (PR prev. using same R)
- -"Perform" a ST
- -Make sure LD data is not "stale"
- -Update the Head ROB pointer

Roll-back if:

- -Brach miss-prediction
- -Stale LD data, LD/ST alias
- -Exceptions: e.g., TLB miss

Rollback

Squash any "newer" instruction
Reset rename table to snapshot (if there is one), or
Rebuild rename table using old PR info in ROB



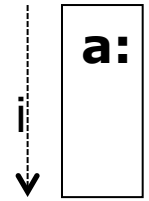
OoO Pipeline Performance:
Four ways to add 2k numbers

Four Ways to add 2k numbers

```
DEF LARGE   2048              //2k
DEF HUGE    2097152           //2M
//size(double) = 8B
//cacheline = 32B; L1 = 32KB
```

/ Loop A /

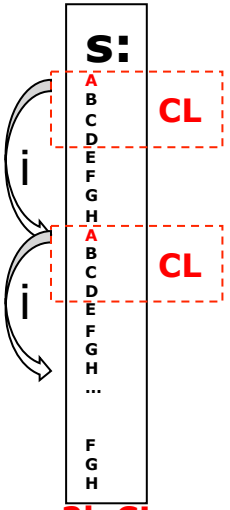
```
double    double[LARGE];
double    sum;
…
for (i =0, i < LARGE, i++) {
   sum+=a[i];
}
```



Uses 512 CL

/ Loop B/

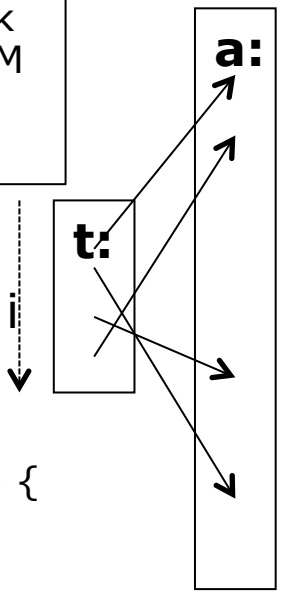
```
struct new_type {double a,b,c,d,e,f,g,h;};
…
new_type  s[LARGE];
double    sum;
…
for (i =0, i < LARGE, i++) {
   sum+=s[i].a;
}
```



Uses 2k CL (every other…)

/ Loop C/

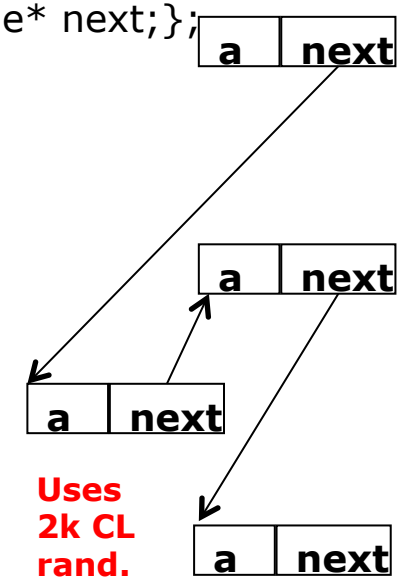
```
double    a[HUGE];
int       t[LARGE];
double    sum;
…
for (i =0, i < LARGE, i++) {
   sum+=a[t[i]];
}
```



Uses 2k rand. +256 CL

/ Loop D /

```
struct node {double a; node* next;};
node* ptr;
ptr=malloc(node);
…
while (ptr->next){
   sum += ptr->a;
   ptr = ptr->next;
}
```



Uses 2k CL rand.

Architectural assumptions

From	To	Latency (bubbles)
FP ALU	FP ALU	1
LD	FP/INT ALU	3 (L1 cache hit)

Branches: Adds 3 cycles, unless correctly predicted

64 bit architecture/application

L1: 32kB 4-way, CL = 32B, latency 3c

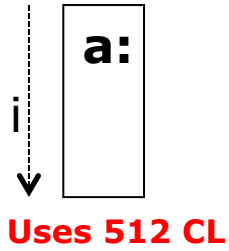
L2: 1MB 4-way, CL = 32B, latency 15c

Memory: Latency 200c

In-order pipeline performance

Loop A

```
double          a[2048];  //size(double)=8B
double          sum;
…
//We assume that this loop is repeated many times//
for (i =0, i < 2048, i++) {
   sum+=a[i];
}
Translated into pseudo-ASM     :
LOOP:
1:      LDD     R4, 0(R1)       // R4 = a[i]  L1 hit, Dep on prev I3
2:      ADDD    R6, R6, R4      // sum +=     Dep on I1; Dep on prev I2
3:      SUBI    R1, R1, #8      // i++        Dep on prev I3
4:      BEQZ    R1, #LOOP       // last time? Dep on I3
```



Uses 512 CL

How many cycles per loop? (on average…)

1. In-order, single-issue, (no BP, no HWP): ??

In-order pipeline performance

Loop A

```
double          a[2048];  //size(double)=8B
double          sum;
…
//We assume that this loop is repeated many times//
for (i =0, i < 2048, i++) {
   sum+=a[i];
}
Translated into pseudo-ASM     :
LOOP:
1:      LDD     R4, 0(R1)       // R4 = a[i]  L1 hit, Dep on prev I3
Stall x3
2:      ADDD    R6, R6, R4      // sum +=   Dep on I1; Dep prev I2
3:      SUBI    R1, R1, #8      // i++      Dep on prev I3
4:      BEQZ    R1, #LOOP       // last time? Dep on I3
Stall x3
```

How many cycles per loop? (on average…)

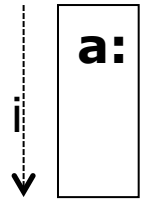
1. In-order, single-issue, (no BP, no HWP): 10c

Four Ways to add 2k numbers

```
DEF LARGE   2048              //2k
DEF HUGE    2097152           //2M
//size(double) = 8B
//cacheline = 32B; L1 = 32KB
```

/ Loop A /

```
double    double[LARGE];
double    sum;
…
for (i =0, i < LARGE, i++) {
   sum+=a[i];
}
```

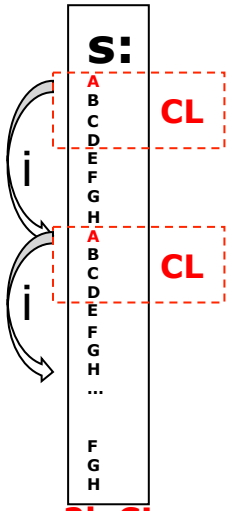


Uses 512 CL

In-order: 10c
OoO: 4c

/ Loop B/

```
struct new_type {double a,b,c,d,e,f,g,h;};
…
new_type  s[LARGE];
double    sum;
…
for (i =0, i < LARGE, i++) {
   sum+=s[i].a;
}
```

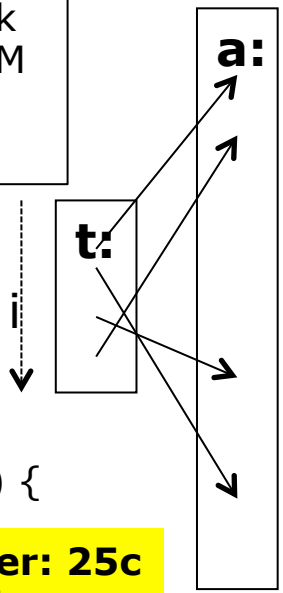


Uses 2k CL
(every other…)

In-order: 22c
OoO: 4c

/ Loop C/

```
double    a[HUGE];
int       t[LARGE];
double    sum;
…
for (i =0, i < LARGE, i++) {
   sum+=a[t[i]];
}
```

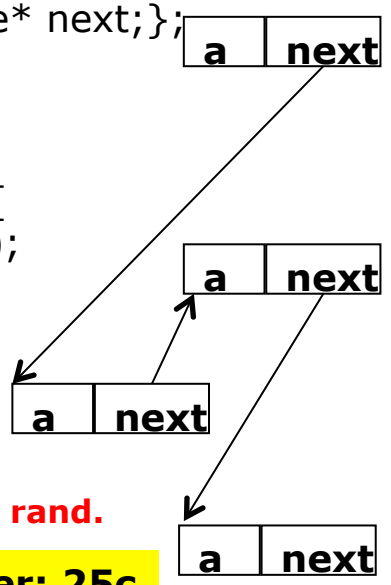


Uses
2k rand.
+256 CL

In-order: 25c
OoO: 5c

/ Loop D /

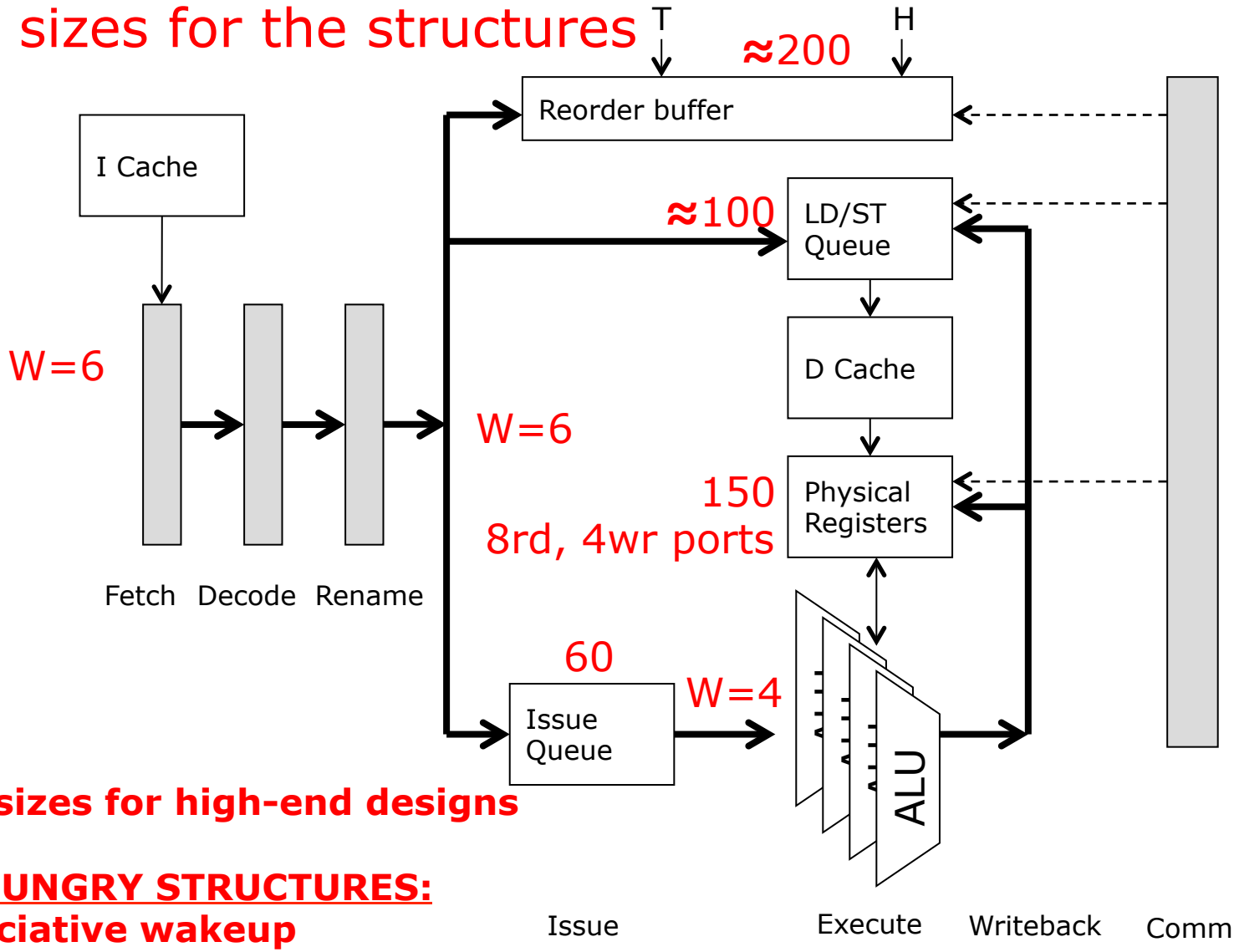
```
struct node {double a; node* next;};
node* ptr;
ptr=malloc(node);
…
for (i =0, i < LARGE, i++) {
   ptr->next = malloc(node);
   ptr = ptr->next;
   ptr-> a = 0
}
…
while (ptr->next){
   sum += ptr->a;
   ptr = ptr->next;
}
```



Uses
2k CL rand.

In-order: 25c
OoO: 22c

Typical sizes for the structures



Common sizes for high-end designs

POWER-HUNGRY STRUCTURES:

IQ – associative wakeup

PRF – large with many ports

LSQ – associative search

FRONT-END – very complex

#MHSRs (Miss-handling status registers)
≈ 16 (max MLP to memory is 16)

OoO in a nutshell

- Can find and explore ILP between different BB
- Can find and explore MLP between BB
- Can hide L2 latency well (if no loop dep.)
- Not enough OoO resources to hide memory accesses
- Costly implementation of some features leads to resource limitations for covering memory latency

How are we doing?

- Create and explore locality:
  - ✓ a) Spatial locality
  - ✓ b) Temporal locality
- Create and explore parallelism
  - ✓ a) Instruction level parallelism (ILP)
  - ✓ b) Thread level parallelism (TLP)
  - ✓ c) Memory level parallelism (MLP)
- Speculative execution
  - ✓ a) Out-of-order execution
  - ✓ b) Branch prediction
  - ✓ c) Prefetching