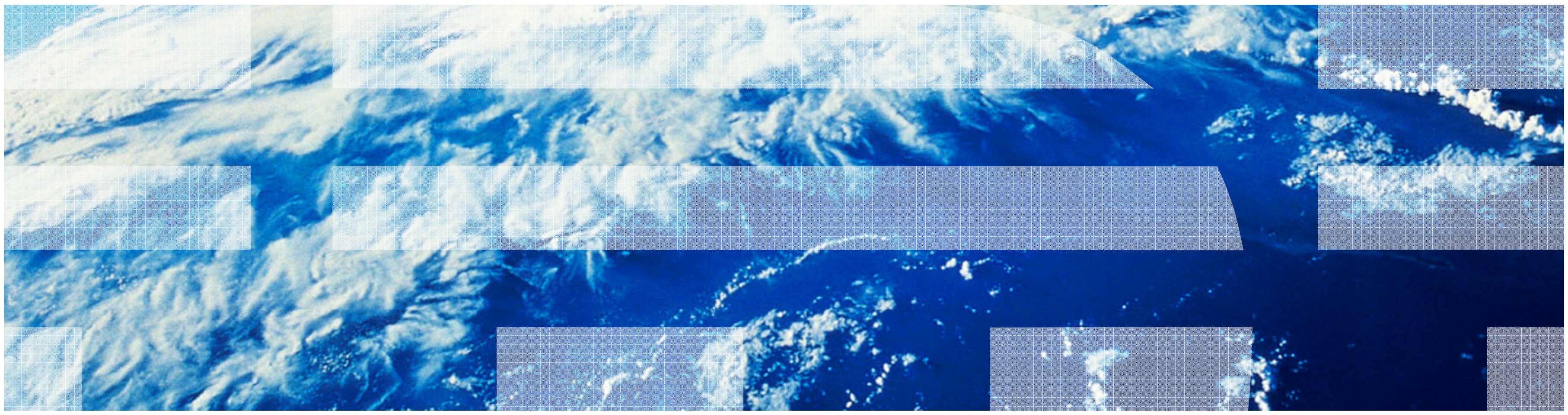
How Do You Do What You Do When You're a z990 CPU?



Trademarks

z/OS
zSeries
z/Architecture
IBM®

Agenda

Introduction

Conceptual View of Instruction Processing

The Cache

Pipeline Architecture

- instruction overlap

Impediments to Instruction Overlap

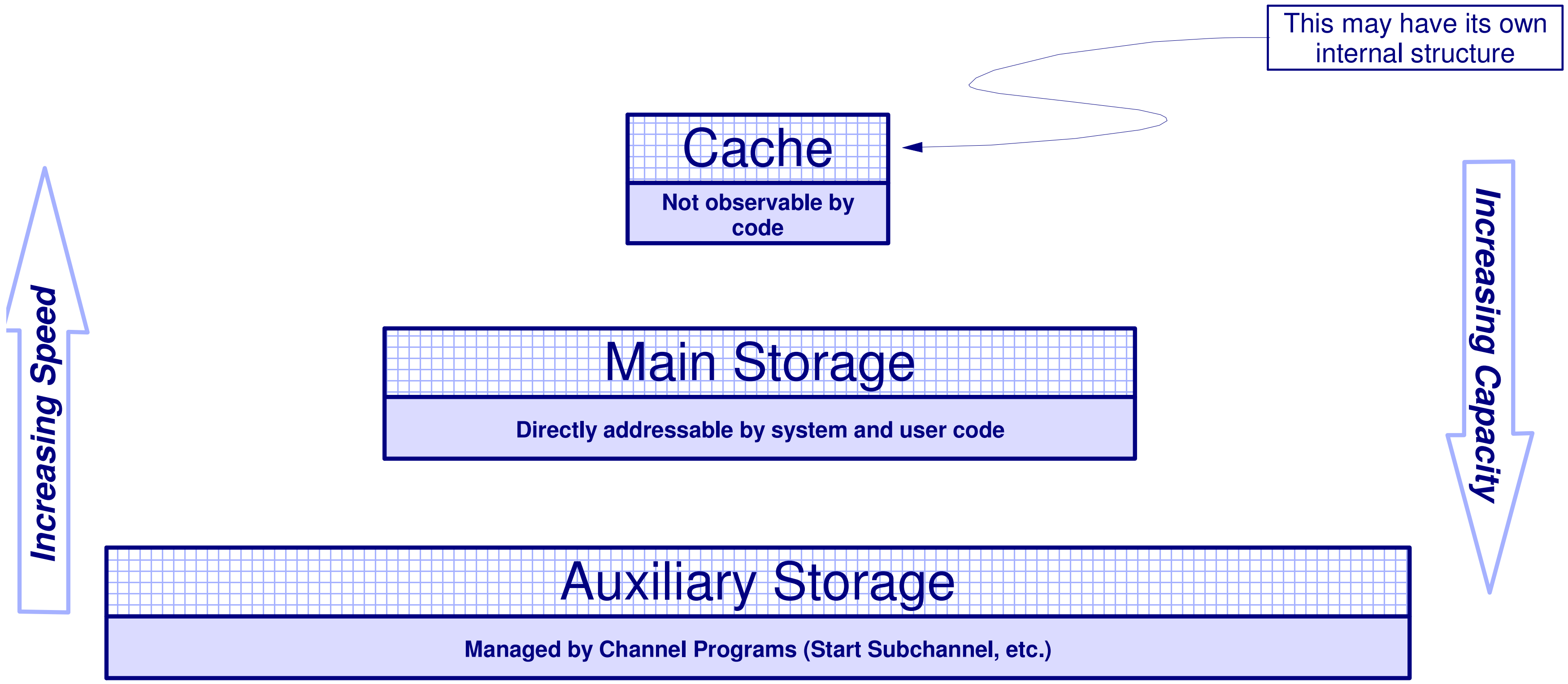
- Address Generation Interlock
- Instruction Fetch Interlock
- Operand Store Compare
- Branches

Superscalar Grouping

- rules and exceptions

The Cache

An element in the hierarchy of system storage.
Managed by the processor, not observable to programs.


This may have its own internal structure
Cache
Not observable by code
Main Storage
Directly addressable by system and user code
Auxiliary Storage
Managed by Channel Programs (Start Subchannel, etc.)
Increasing Speed
Increasing Capacity

Cache Effects

Cache is to main storage as main storage is to the backing store in the paging subsystem

- Therefore, many of the concepts we are familiar with from "virtual storage" are applicable when thinking about cache.

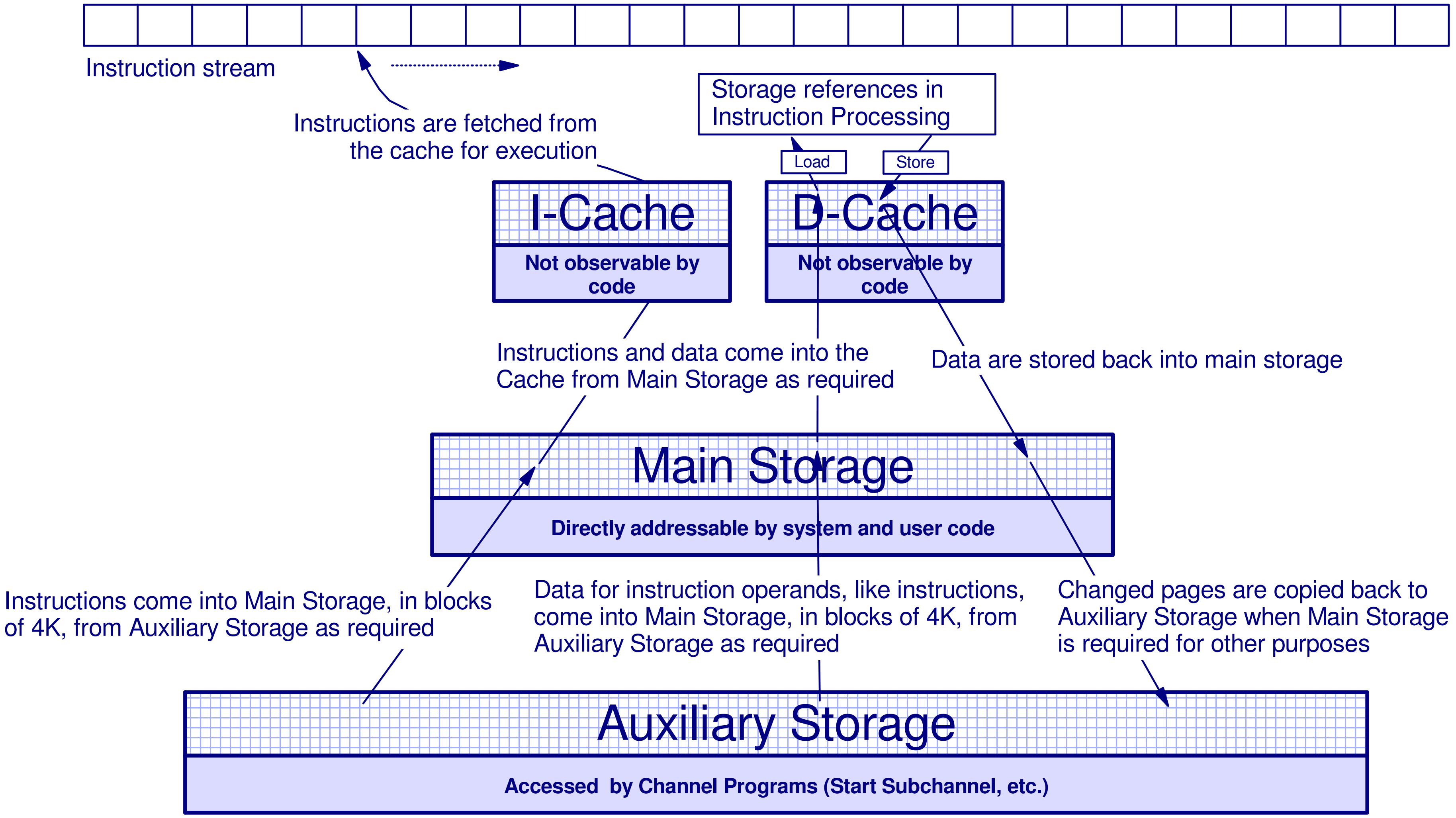
The concepts of “thrashing”, “working set”, and “locality of reference” that are normally applied to Virtual Storage apply to the cache as well:

- where possible, address data sequentially rather than randomly
- keep data referenced closely in time close in memory
- keep code compact, avoid branches

Cache is completely under control of the processor

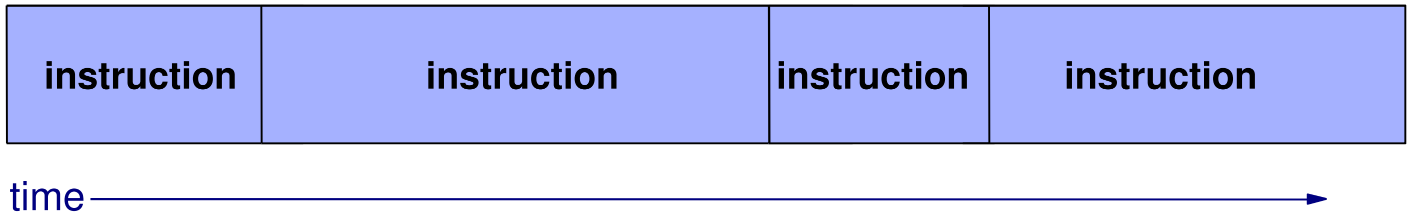
- in a multiprogramming system there’s nothing you can do to make its behavior deterministic.

Instruction Processing and the Cache


Instruction stream
Instructions are fetched from the cache for execution
Storage references in Instruction Processing
Load
Store
I-Cache
Not observable by code
D-Cache
Not observable by code
Instructions and data come into the Cache from Main Storage as required
Data are stored back into main storage
Main Storage
Directly addressable by system and user code
Instructions come into Main Storage, in blocks of 4K, from Auxiliary Storage as required
Data for instruction operands, like instructions, come into Main Storage, in blocks of 4K, from Auxiliary Storage as required
Changed pages are copied back to Auxiliary Storage when Main Storage is required for other purposes
Auxiliary Storage
Accessed by Channel Programs (Start Subchannel, etc.)

Conceptual View of Execution

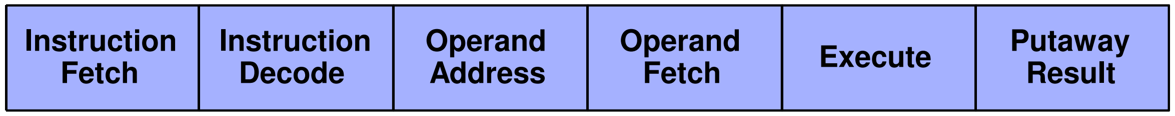
Instructions are executed in the order they are seen.
Every instrucion completes before the following instruction begins.



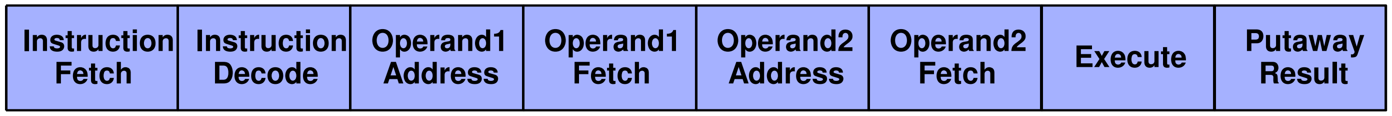
Instructions take a varying amount of time.
Instructions have direct and immediate access to main storage.

Pipeline View of Instructions

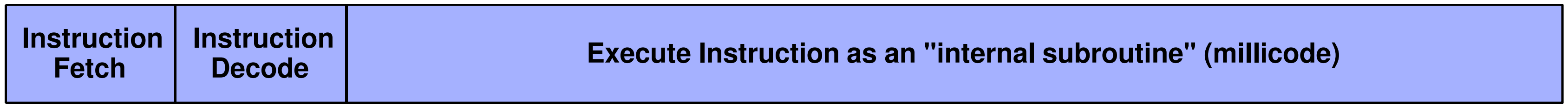
Individual instructions are really a sequence of dependent activities, varying by instruction:



for example: `A   R1,D2(X2,B2)`



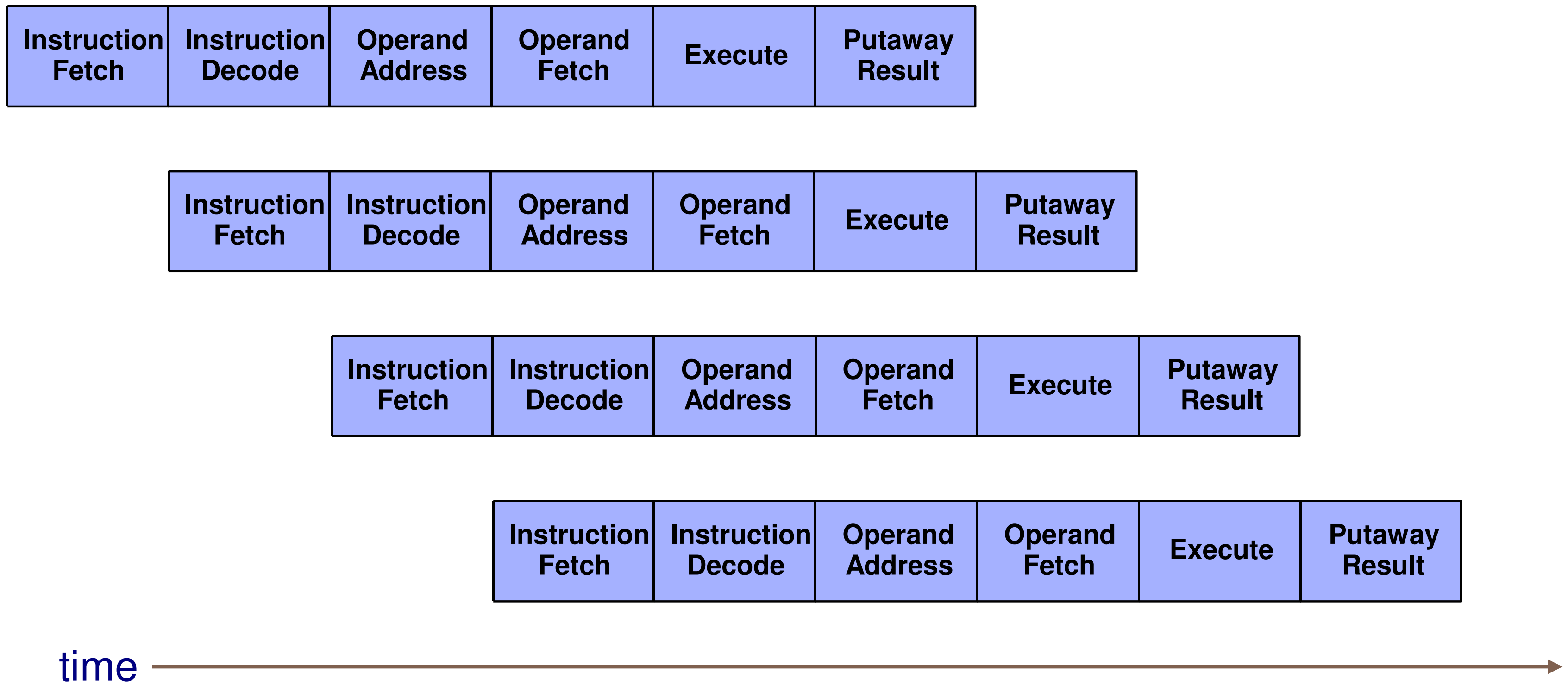
for example: `CLC   D1(L,B1),D2(B2)`



for example: `UPT` **(Update Tree)**

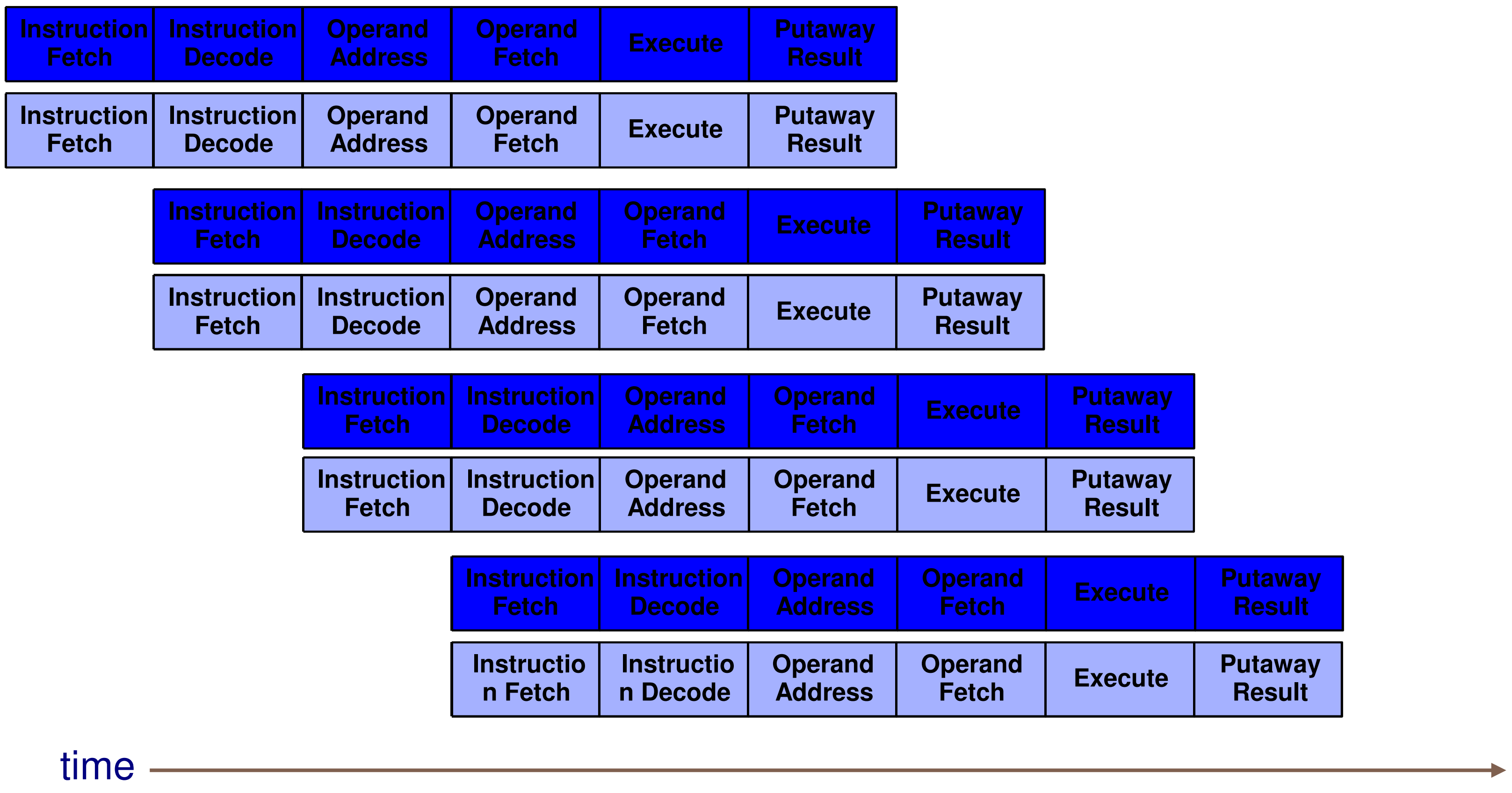
Overlapped Execution in a Pipeline

If each stage in the execution of an instruction is implemented by distinct components then execution can be overlapped.



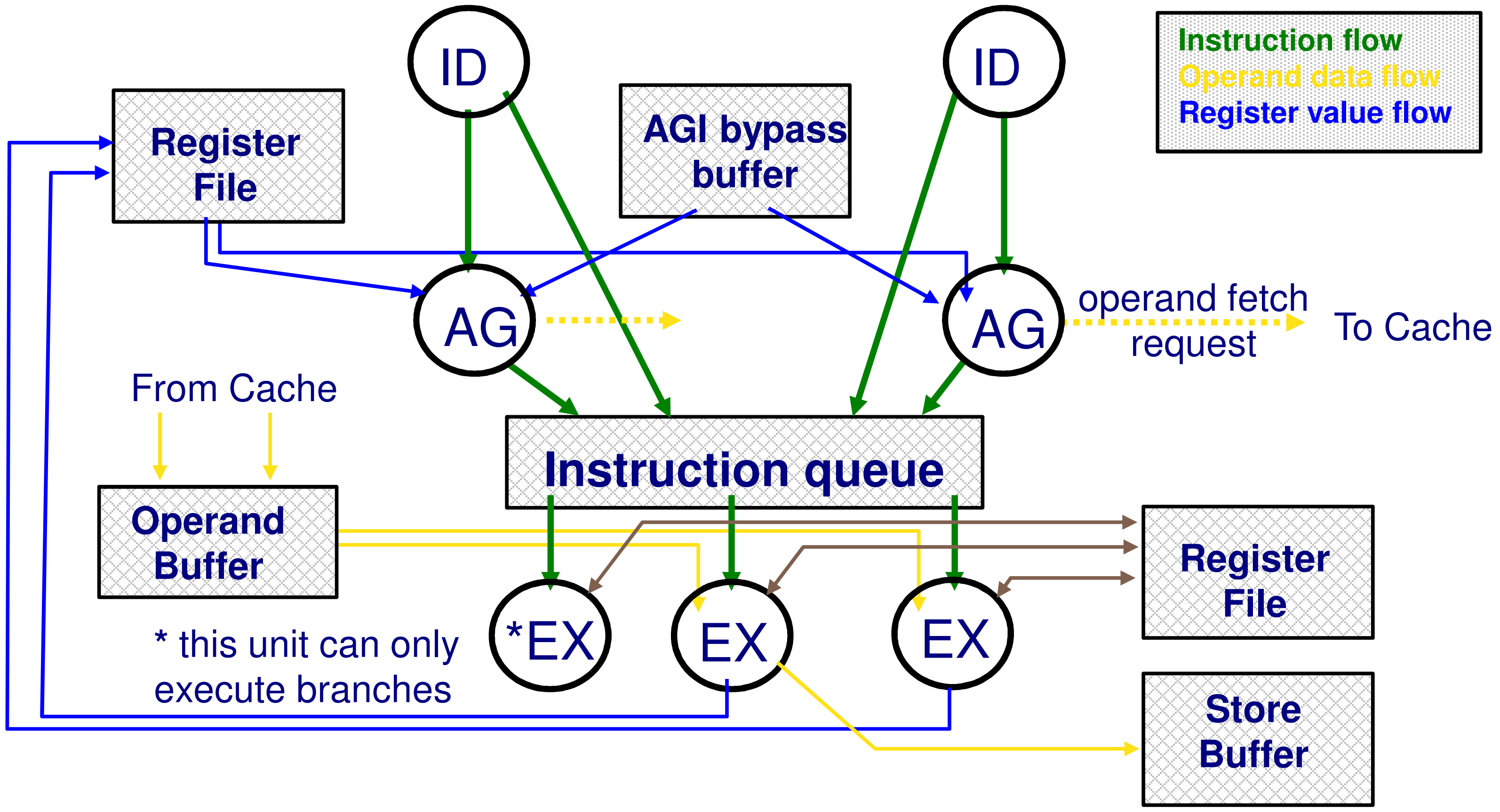
Superscalar multiple instruction overlap

A Superscalar processor can process multiple instructions simultaneously because it has multiple units for each stage of the pipeline. But, the apparent order of execution is still maintained.



Superscalar Instruction Pipeline

Schematic of superscalar, in-order execution, out-of-order operand fetch instruction pipeline (based on z990).



Impediments to Instruction Overlap

Address Generation Interlock

- waiting for the results of a previous instruction to compute an operand address

Instruction Fetch Interlock

- reloading instructions as a result of stores into the instruction stream

Operand Store Compare

- waiting to refetch a recently modified operand

Branch Misprediction

- branching (or not branching) in a way other than the processor has guessed.

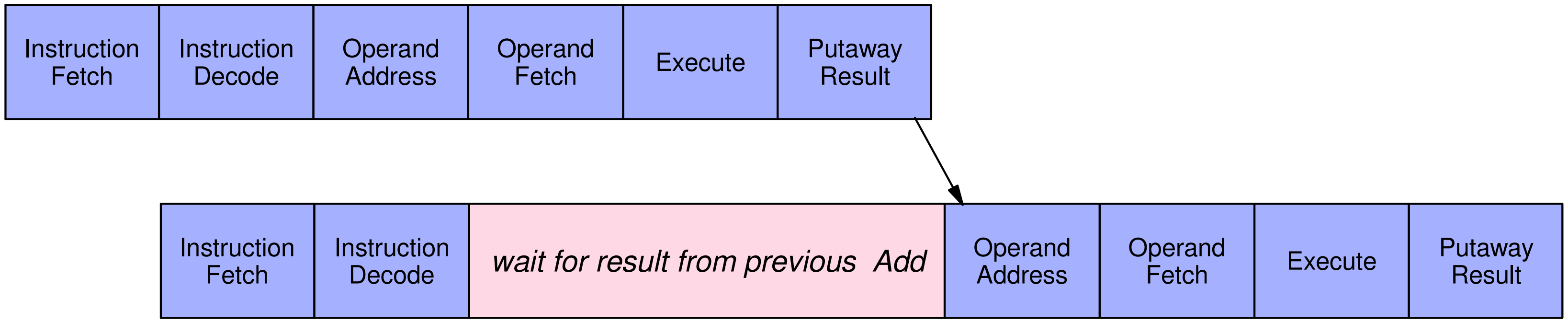
Inhibition of Superscalar Grouping

- executing less than the optimal number of instructions simultaneously due to inter-instruction dependencies

Address Generation Interlock

Occurs when Operand Addressing in one instruction has to wait for the result of a previous instruction:

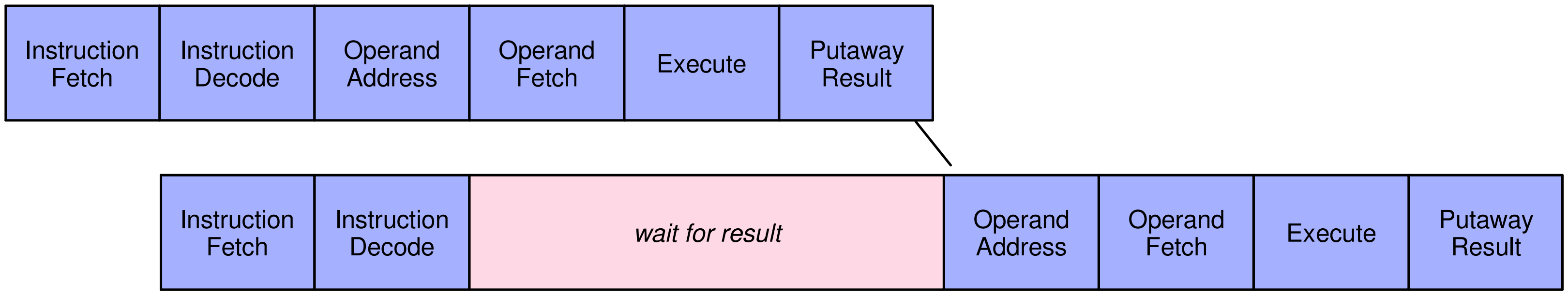
```
A    R1,Stride
L    R2,Vector(R1)
```



AGI Gap

The size of the “wait for result” gap depends on the structure and complexity of the pipeline.

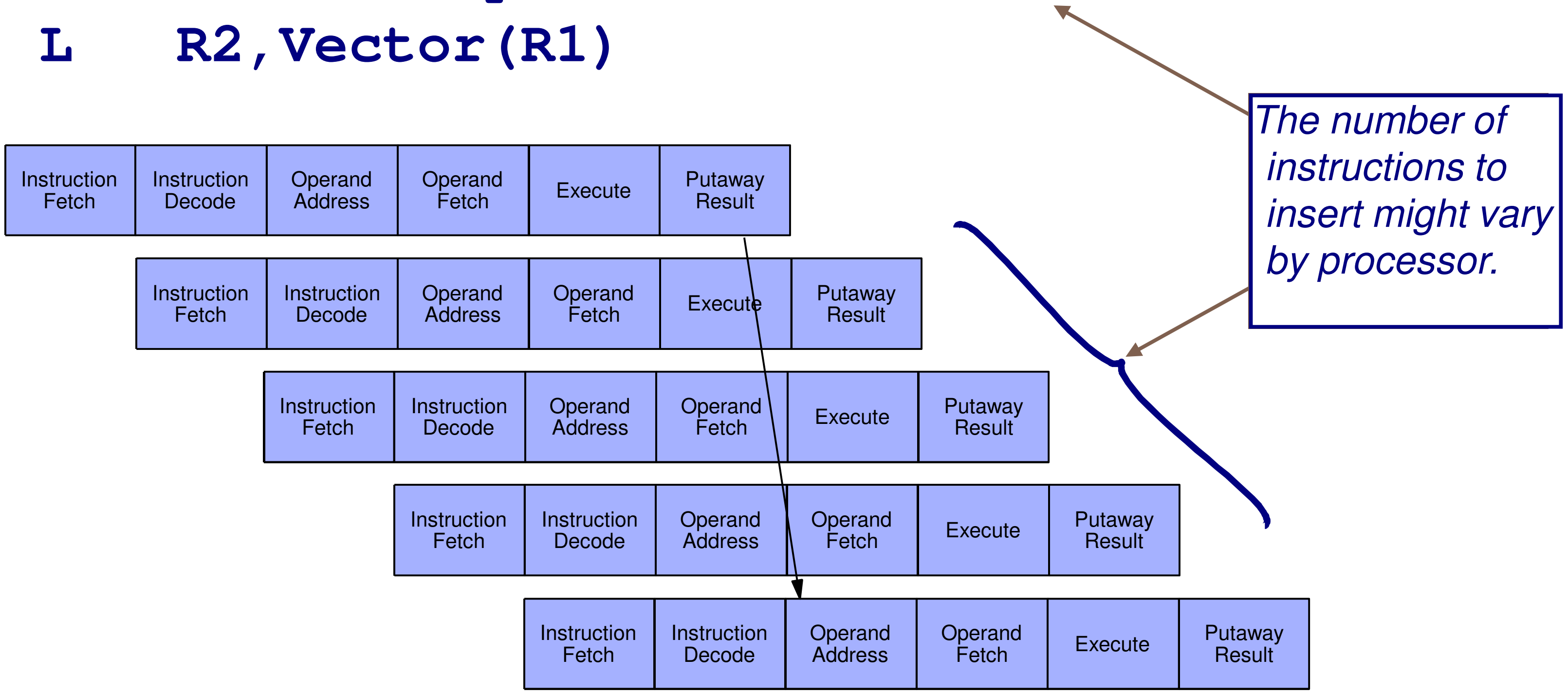
- The gap can be reduced in software by careful ordering of the instructions.
- It can also be reduced in hardware by an engineering trick called "AGI bypass".



Address Generation Interlock (continued)

This sequence could be replaced, at little or no cost to execution time with:

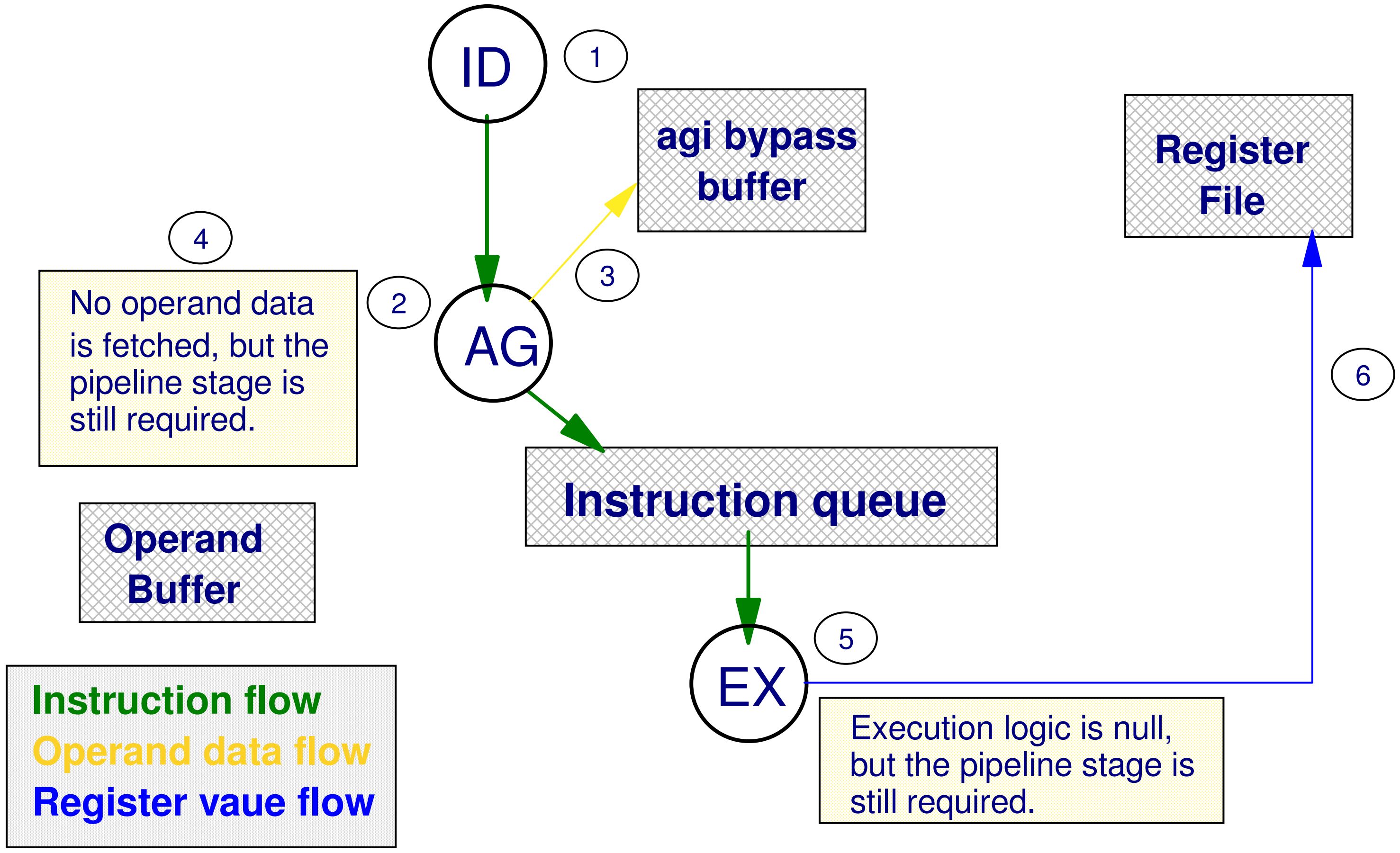
```
    A    R1,Stride
*   Unrelated simple instructions
    L    R2,Vector(R1)
```



Note that this in not a superscalar example.

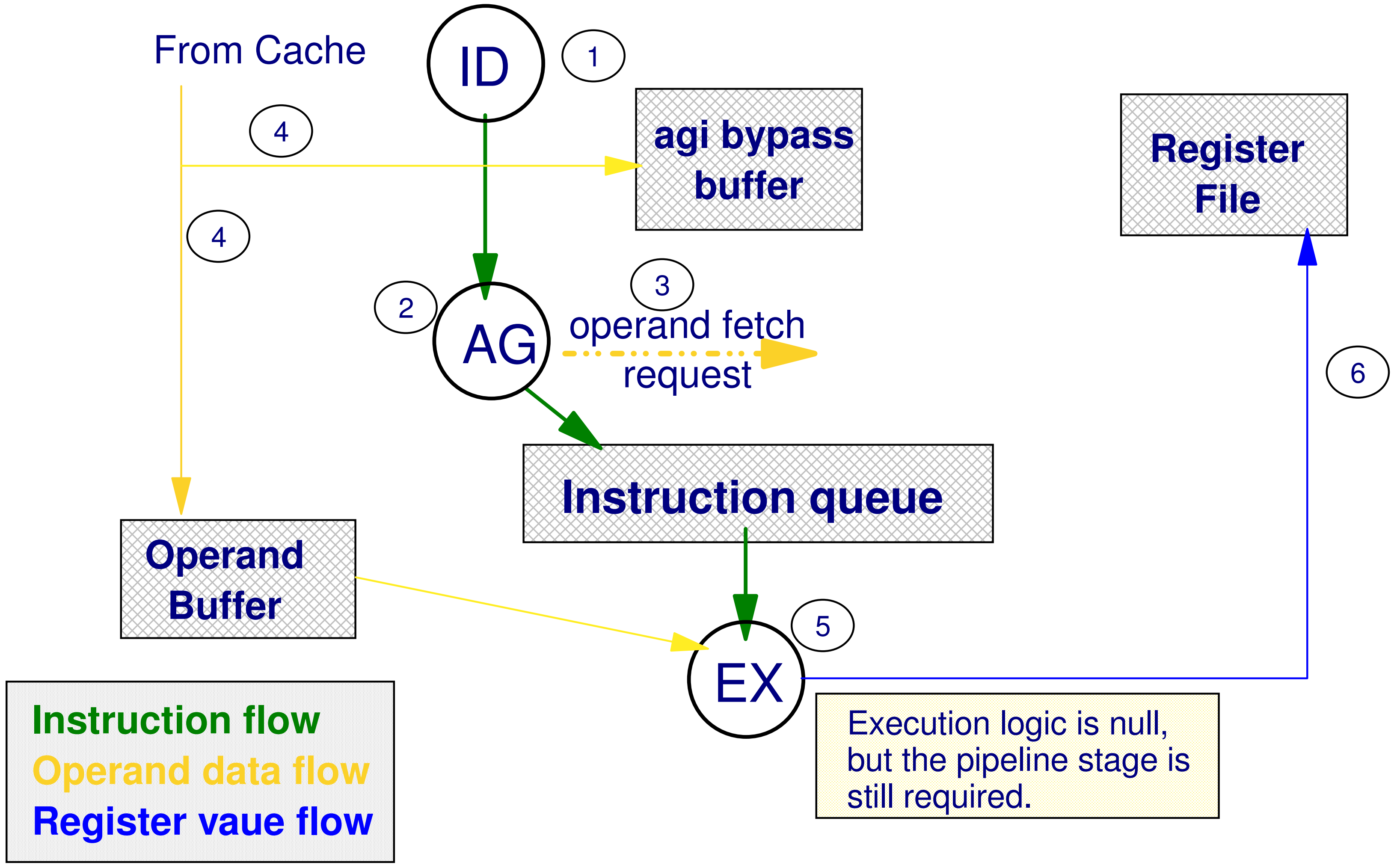
Data flow for Load Address AGI bypass

Updated register value is available after step 3 rather than 6.


ID
1
agi bypass buffer
Register File
4
No operand data is fetched, but the pipeline stage is still required.
2
AG
3
6
Instruction queue
Operand Buffer
5
EX
Instruction flow
Operand data flow
Register vaue flow
Execution logic is null, but the pipeline stage is still required.

Data flow for Load AGI bypass

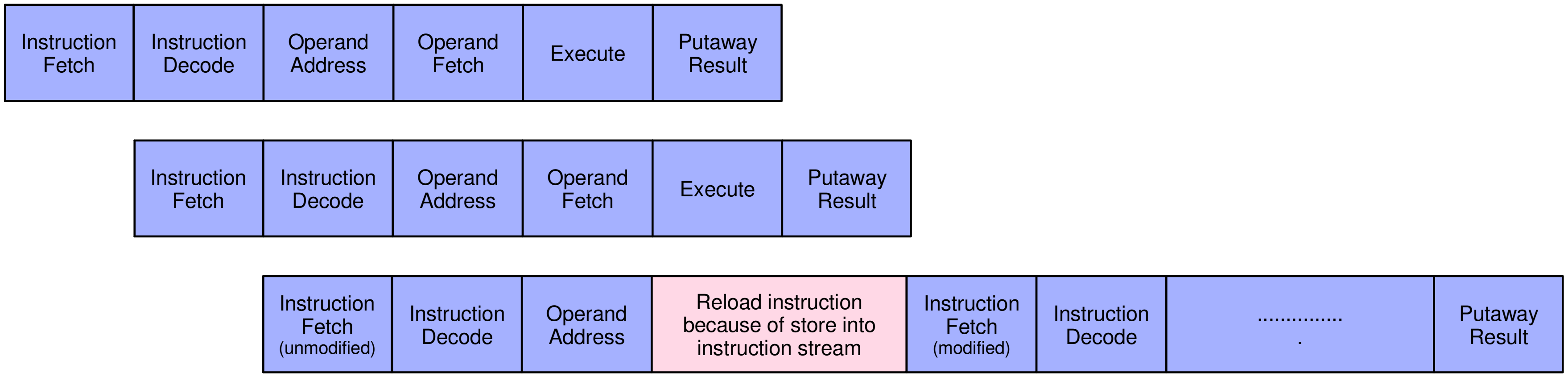
Updated register value is available after step 4 rather than 6.



Instruction Fetch Interlock

Occurs when an instruction is (*or might be*) modified by a store in a previous instruction

```
   LH   R3,BranchTable(R2) S-type adcon
   STH  R3,BranchTarget
   B    *-*                Modified Instruction
BranchTarget  EQU  *-2
```



Note that this is a unified cache example.

Instruction Fetch Interlock: Notes

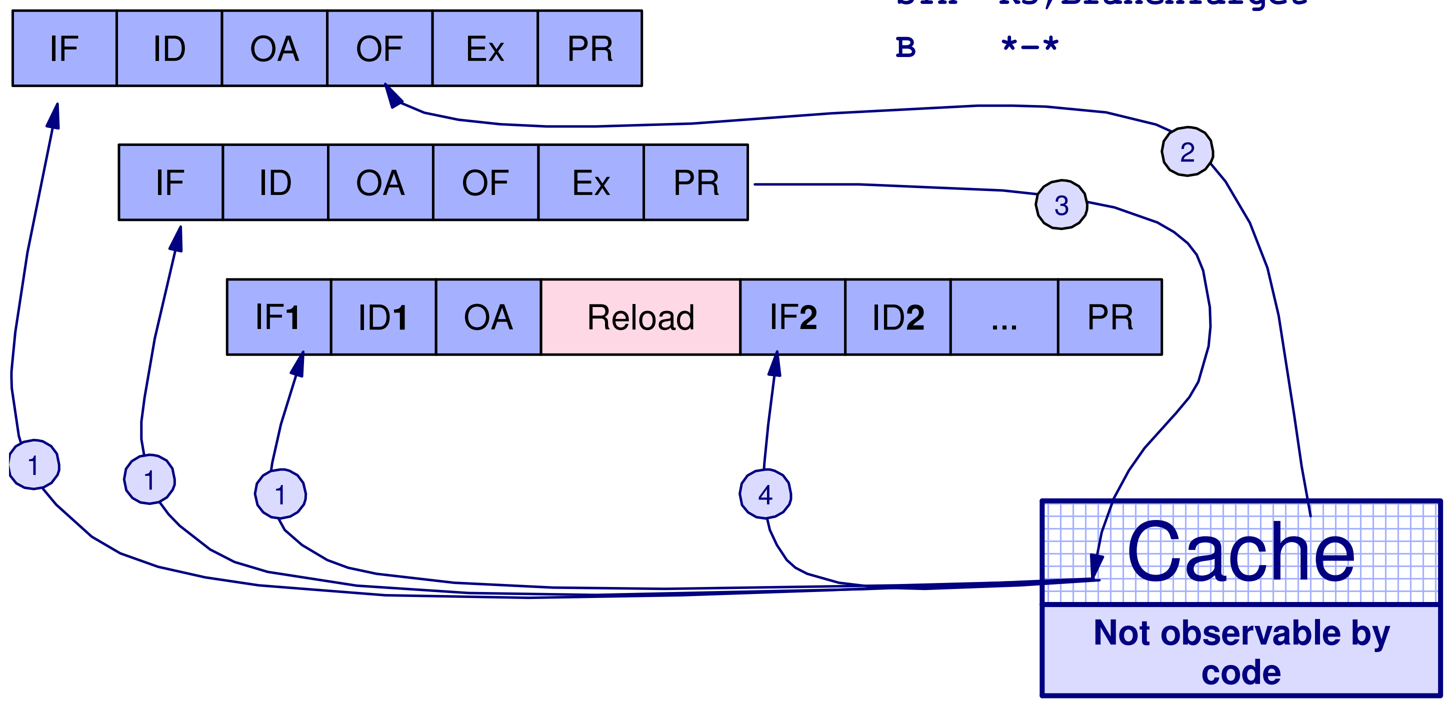
Reentrant code does not suffer from Instruction Fetch Interlock

There are cases where stores into the immediate instruction stream *might* not be recognized by the processor:

- Access-Register mode
- Home-Space mode
- When stores are made to a real address using a different effective address
- These are discussed in detail in *Principles of Operation*, "5.13.1 Conceptual Sequence" at
  `publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/DZ9ZR000/5.13.1`

IFI, The Unified Cache View

```
LH    R3,BranchTable(R2) S-type adcon
STH   R3,BranchTarget
B     *-*
```



1. Fetch instructions from the cache, possibly going to main storage in the first instance
2. Fetch operand from the cache, possibly delayed as cache is filled from main storage
3. Store result in the cache.
4. The next instruction to be executed has already been fetched (IF1) from the same cache block as the previous store; this fetch is invalidated and must be redone (IF2)

The Split Cache

On zSeries processors, the Cache has an internal hierarchy:

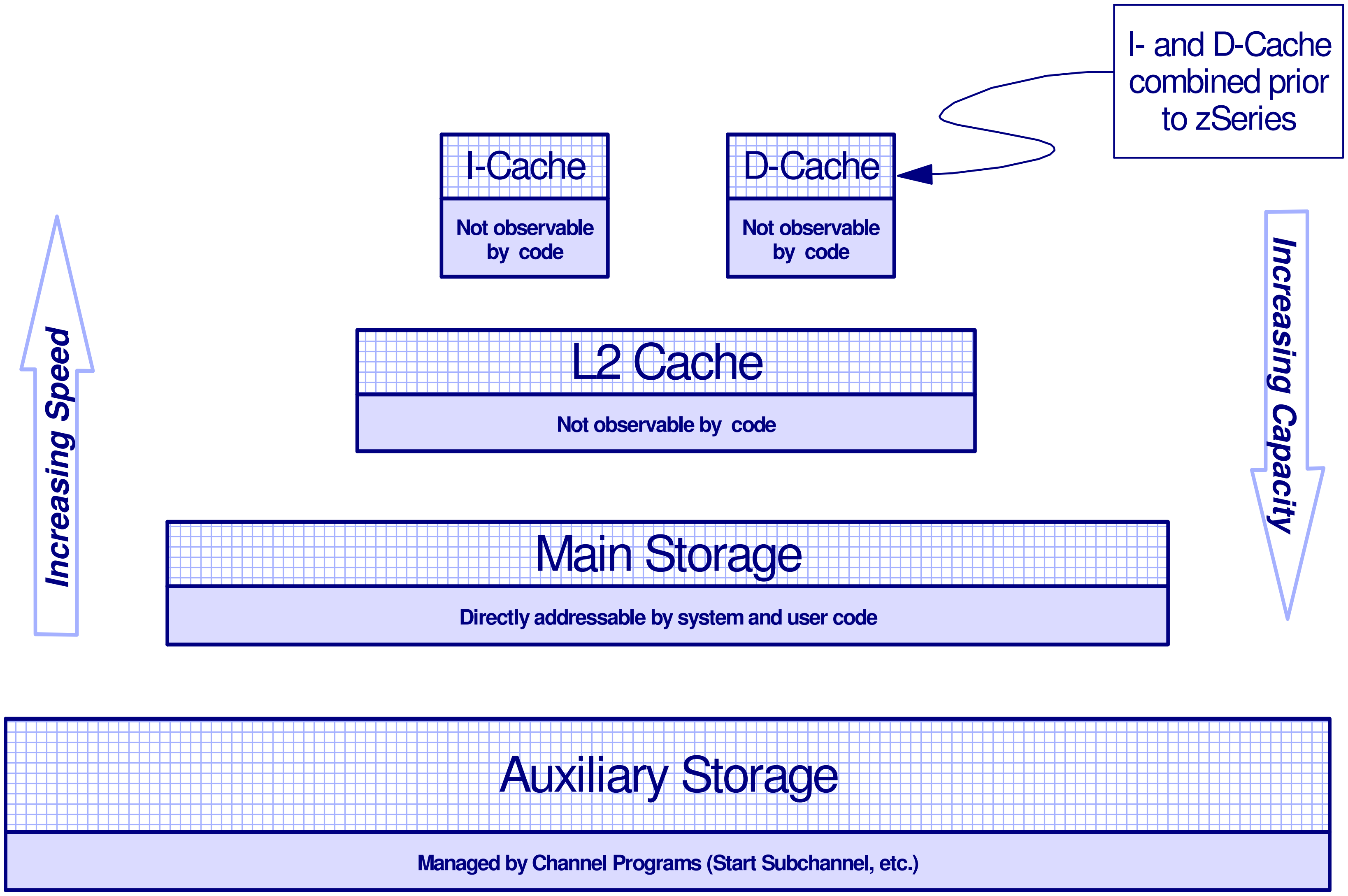
- Level-1 Cache (closest to the processor is the storage hierarchy) and
- Level-2 Cache, between the Level-1 Cache and Main Storage

On zSeries processors, the L-1 Cache is split between an Instruction Cache (I-Cache) and a Data Cache (D-Cache)

This structure allows caches that:

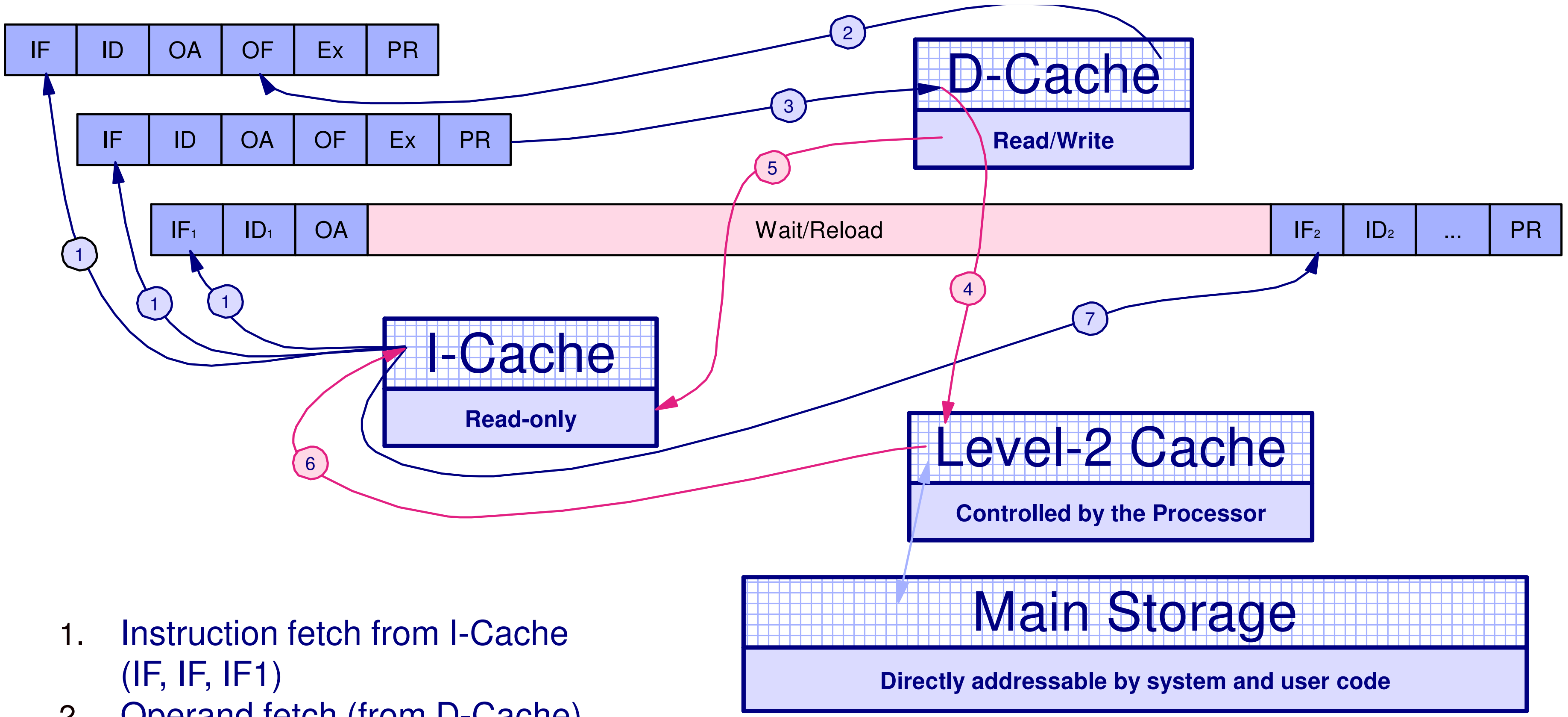
- are larger, supporting a larger working set
- have more opportunities for parallelism among processor components
- are faster

The Cache, Decomposed



The Split Cache View of IFI

Effect of Instruction Cache/Data Cache split



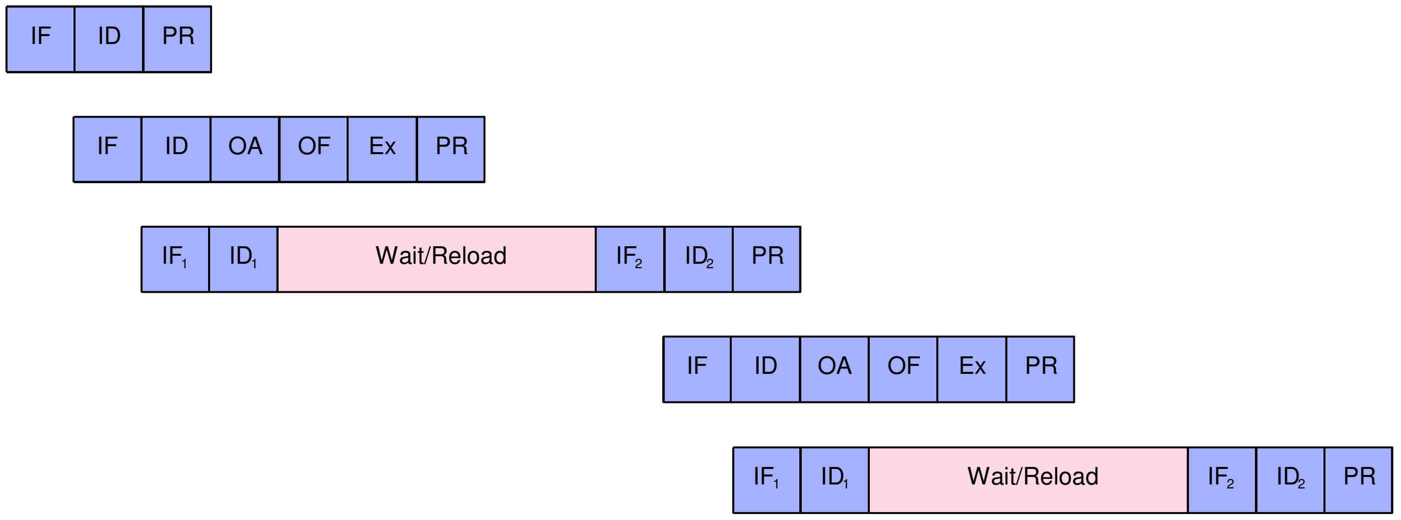
1. Instruction fetch from I-Cache (IF, IF, IF1)
2. Operand fetch (from D-Cache)
3. Store result in D-Cache
4. Push Result through to Level-2 Cache
5. Invalidate I-Cache in the same 256-byte block
6. Reload I-Cache from L-2 Cache
7. Refetch potentially-changed instruction from I-Cache (IF2)

IFI, Another Example

Consider the following macro expansion

```
                         9      load   eploc=(R2),loadpt=(R3)
                        12+     CNOP   0,4
4110 xxxx   xxxxx       13+     LA     1,*+8        PICK UP ADDR OF PARM LIST
47F0 xxxx   xxxxx       14+     B      *+20         BRANCH AROUND CONSTANTS
00000000                15+     DC     A(0)         EPLOC OR DE PARAMETER
00000000                16+     DC     AL4(0)       DCB ADDRESS PARAMETER
00                      17+     DC     AL1(0)       PARAMETER LIST FORMAT NUMBER
00                      18+     DC     AL1(0)       RESERVED
01                      19+     DC     B'00000001'
00                      20+     DC     B'00000000'  OPTIONS
00000000                21+     DC     A(0)         EXPLICIT LOAD OR LOADPT
4100 2000   00000       22+     LA     0,0(0,R2)    PICKUP EPLOC OR DE PARM
5000 1000   00000       23+     ST     0,0(0,1)     STORE IN SUPV PARM LIST
4100 3000   00000       24+     LA     0,0(0,R3)    PICKUP LOADPT ADDRESS
5000 100C   0000C       25+     ST     0,12(0,1)    STORE LOADPT ADDRESS
41F0 0009   00009       26+     LA     15,9         LOAD EXTENDED SVC ROUTING
                                                    CODE
0A7A                    27+     SVC    122          ISSUE EXTENDED SVC
```

IFI, Another Example *(continued)*



```
LA    0,0(0,2)
ST    0,0(0,1)
LA    0,0(0,3)
ST    0,12(0,1)
LA    15,9
```

Note that the 2 Store instructions do not modify instructions but, because they store into the nearby stream, subsequent instructions must be reloaded.

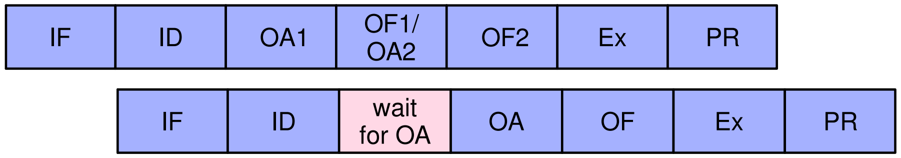
- The granularity for detecting potential instruction fetch interlock is the cache line (256 bytes).

Operand Store Compare (OSC)

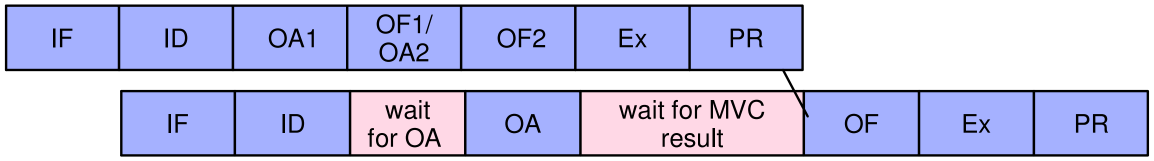
Occurs when an instruction requires an operand from storage that is unavailable because it will be modified by a previous instruction

- called Operand Store Compare because it often happens when testing a recently derived result
- The granularity of checking for an interlock is a doubleword
- There may also be an additional delay until the operand value has "settled down".

```
MVC   WorkArea(8),Uservar
CLI   WorkArea+7,C' '
```



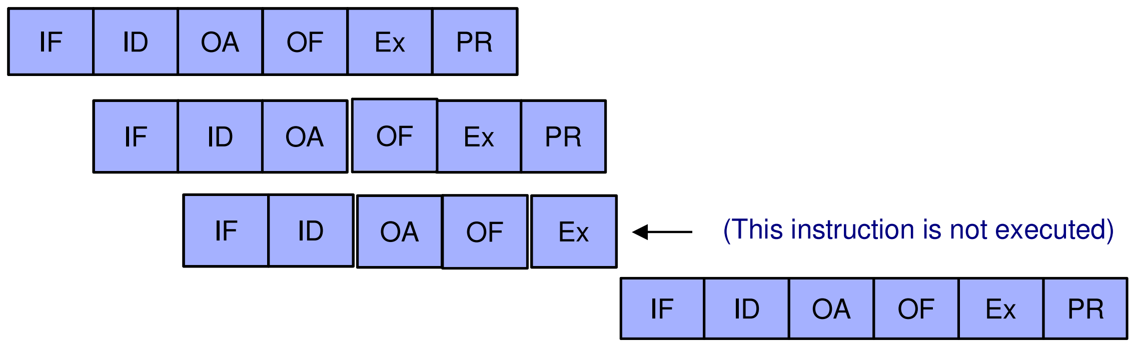
```
MVC   WorkArea(8),Uservar
CLI   Uservar+7,C' '
```



Branches and the Branch History Table

In order to keep the pipeline supplied with instructions, the processor must "guess" the next instruction after a branch.

- The processor maintains a branch history table (BHT) of taken branches. Branches not taken do not take up space in the BHT.
- If there is no BHT entry, the processor initially assumes the branch is not taken (except for BC 15, BCT, BXLE).
- A mispredicted branch causes a partial flush of the pipeline.



```
   TM    Flg,Bad
   BZ    OK
*  error path
OK L     R1,...
```

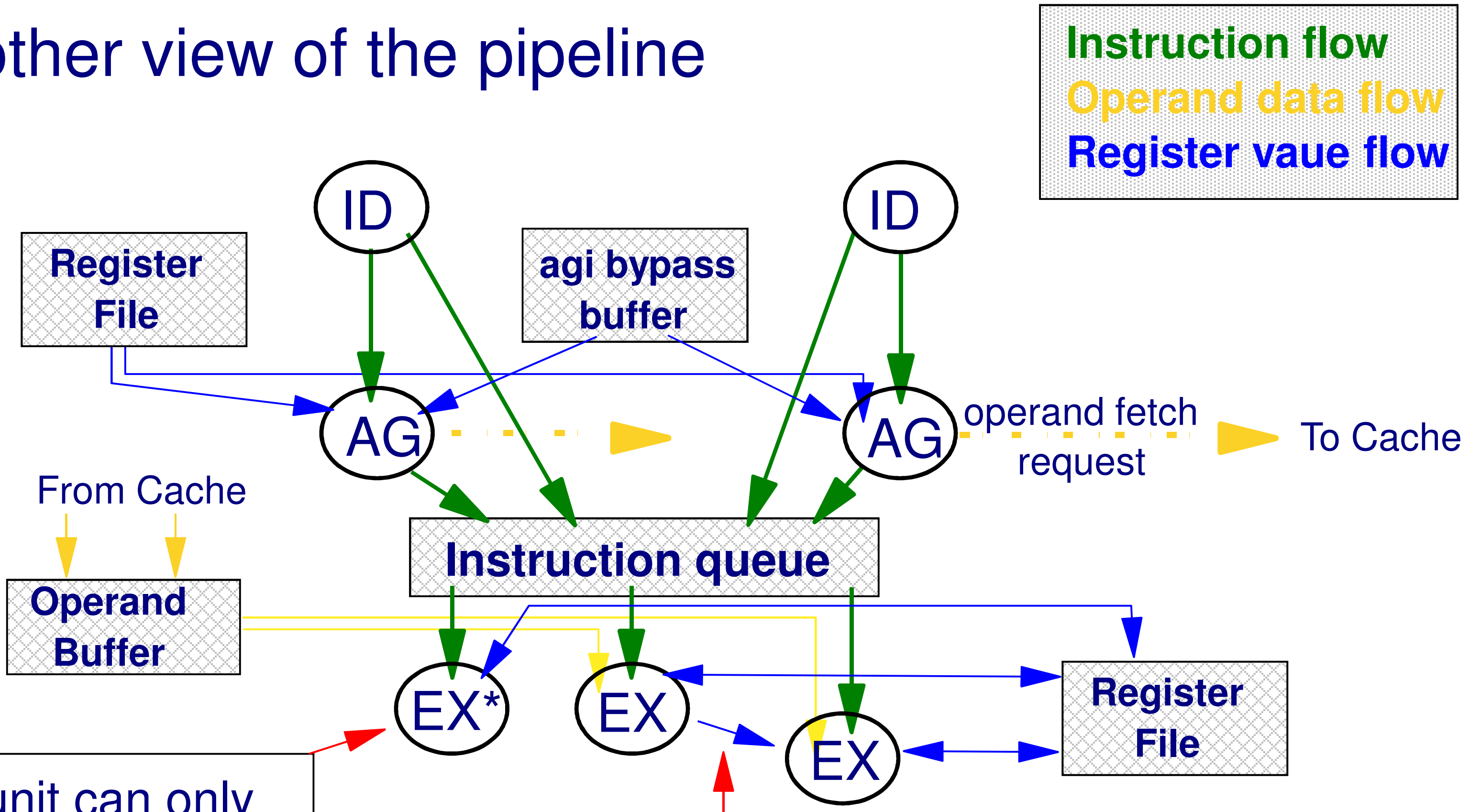
Superscalar Grouping Rules

The z990 has 3 execution units and is capable of executing up to 3 instructions simultaneously.
A sequence of up to 3 instructions can be selected for execution subject to the following basic rules (exceptions to follow):

- The first execution unit can only execute branch instructions and branch instructions can only be executed by this unit.
- Multi-cycle (and some single-cycle) instruction must execute alone as a group of one instruction
- Only one store instruction is allowed per group
- An instruction which references a register cannot be grouped with a preceding instruction that sets the register.

Superscalar Grouping Pipeline

Another view of the pipeline



This unit can only execute branch instructions and they can only execute in this unit

Execution of the third unit is staggered so that it can sometimes take as input an output of the second unit.

Superscalar Grouping Exceptions

Clever engineering tricks allow some grouping otherwise prohibited by the basic rules.

- Operand Forwarding: Some instructions[1] can "forward" a register they update to some other instructions[2] and they can be grouped. But, an instruction that sets only the low 32 bits cannot forward to a 64-bit instruction.
- Condition Code Forwarding: Some instructions which set a register and the condition code and "forward" the CC setting to LTR which then sets the CC based on how it was set by the previous instruction.
- LTR Special: LTR of a register into itself is not considered to set that register and can be grouped with a following instruction that references the register.

*1 e.g. L, LR, LTR, LG, LGR, LGTR, LA, LARL
*2 e.g. A, AR, LPR, LCR, S, SR, AG, AGR, LPGR, LCGR, SG, SGR

Code Reordering Example

Original Code Sequence

7 instruction groups and 10 cycles AGI delay

```
AGI seq     instruction text                | seq   instruction text
    01   LLGT   @04,XFORNP31                 |
<4> 02   L      @04,FW(,@04)                 | 03   ST     @04,XFORS
    04   LG     @05,TOPPTR                   |
<2> 05   LG     @09,RTTOP(,@05)              |
<2> 06   ST     @04,RSISIZE(,@09)            | 07   SLR    @02,@02
    08   ST     @02,RSIPREV(,@09)            | 09   LG     @02,RDIPTR64
<2> 10   LH     @08,RDITYPE(,@02)            |
```

Reordered Code Sequence

5 instruction groups and 6 cycles AGI delay

```
AGI seq     instruction text                | seq   instruction text
    01   LLGT   @04,XFORNP31                 | 04   LG     @05,TOPPTR
<2> 05   LG     @09,RTTOP(,@05)              | 07   SLR    @02,@02
<2> 02   L      @04,FW(,@04)                 | 06   ST     @04,RSISIZE(,@09)
    08   ST     @02,RSIPREV(,@09)            | 09   LG     @02,RDIPTR64

<2> 03   ST     @04,XFORS                    | 10   LH     @08,RDITYPE(,@02)
```