Chapter 3

Instructor: Josep Torrellas

CS433

Instruction Level Parallelism (ILP)

- Would like to exploit the independence of instructions in order to allow overlap of these instructions in the pipeline
- Amount of parallelism among instructions may be small - need ways to exploit the parallelism within the code
- Can substantially reduce the amount of work that is needed to run the code

Potential Drawback

- From the last chapter we know :
- Pipeline CPI = Ideal Pipeline CPI + Structural Stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls
- With the simple pipeline only concerned with RAW and control stalls, advanced techniques make WAR stalls and WAW stalls new concerns that must be dealt with

Obstacles to ILP - Dependences

- Data Dependence - due to one instruction producing the result needed by another instruction

```
ie            L5:   LD      F4, 0(R3)
                    ADDD  F6,F4,F2   ; data dependent on LD
                    SD      0(R3),F6   ; data dependent on ADDD
```

- Name Dependence - due to two instructions using the same register or memory location, without the flow of data between the instructions.

```
ie  1  L5:  LD       F4,O(R3)
    2       ADDD  F6,F4,F2     ; data dependent on 1
    3       SD       O(R3),F6    ; data dependent on 2
    4       LD       F4,-8(R3)    ; name dependent on 1+2
    5       ADDD   F6,F4,F2     ; data dependent on 4, name dependent on 2+3
    6       SD       -8(R3),F6    ; data dependent on 5
    7       SUBI    R1,R1,#16
    8       BNEZ    R1,L5         ; data dependent on 7
```

- Antidependence - corresponds to a WAR hazard - instructions i + c writes a register that instruction i reads
- Output Dependence - corresponds to a WAW hazard - instruction i and i + c write the same register or memory location
- Control Dependence - ordering of instructions must be determined so that a non-branch instruction only executes when it should - due to branches

ie
```
 1  L5:  LD      F4,0(R3)
 2       ADDD    F6,F4,F2
 3       SD      0(R3),F6
 4       SUBI    R1,R1,#8
 5       BNEZ    R1,exit
 6       LD      F4,0(R3)     ; control dependent on 5
 7       ADDD    F6,F4,F2     ; control dependent on 5
 8       SD      0(R3),F6     ; control dependent on 5
 9       SUBI    R1,R1,#8     ; control dependent on 5
10       BNEZ    R1,L5        ; control dependent on 5
    exit :
```

Eliminating Dependencies

- Antidependences - Register renaming - can be static or dynamic - simply use different registers for each “body” of code

```
ie  1  L5:  LD      F4,0(R3)
    2       ADDD    F6,F4,F2
    3       SD      0(R3),F6      ; end of body 1
    4       LD      F9,-8(R3)
    5       ADDD    F11,F9,F7
    6       SD      -8(R3),F11    ; end of body 2
    7       SUBI    R1,R1,#16
    8       BNEZ    R1,L5
```

Eliminating Dependencies

- Control dependences -eliminate intermediate branches

```
ie 1   L5:   LD      F4,0(R3)        L5:   LD  F4,0(R3)
   2         ADDD F6,F4,F2                 ADDD F6,F4,F2
   3         SD       0(R3),F6             SD       0(R3),F6
   4         SUBI    R1,R1,#8
   5         BNEZ    R1,L5                 LD       F4,0(R3)
   6         LD       F4,0(R3)             ADDD F6,F4,F2
   7         ADDD   F6,F4,F2               SD       0(R3),F6
   8         SD       0(R3),F6             SUBI    R1,R1,#16
   9         SUBI    R1,R1,#8              BNEZ   R1,L5
   10        BNEZ   R1,L5
```

+ Also reduces static and dynamic IC

- total iterations must be a multiple of number of "bodies"

- What about data dependences ?

  We usually dont eliminate Data dependences - we try to avoid Data Dependences - ie scheduling

Dynamic Scheduling

- Static Scheduling : if there is a hazard, stop the issue of the instruction and the ones that follow
- Dynamic Scheduling : hardware rearranges the exec of instructions to reduce stalls

+ handles cases when dependences are unknown at compile time (e.g. involve a mem. ref.)

+ simplify compiler

+ allows code compiled for 1 pipeline to run on another

- significant hardware complexity

Why Dynamic Scheduling ?

DIVD F0,F2,F4

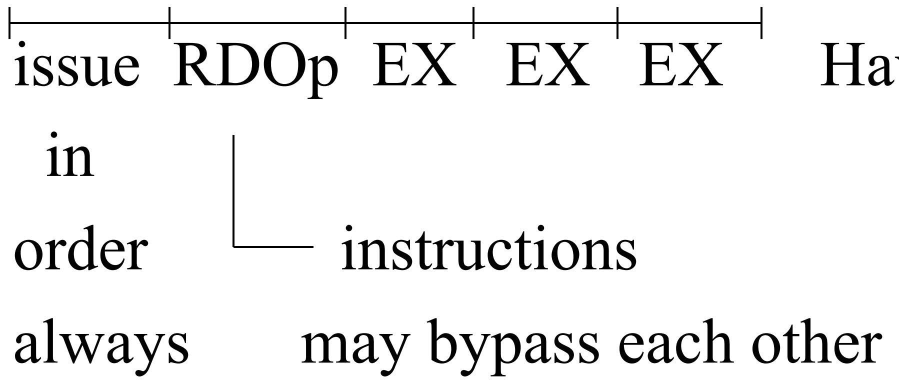
ADDD F10,F0,F8

SUBD F12,F8,F14 ← stuck, even though not dependent

- After the instruction fetch:
  - Check structural hazards
  - Wait for the absence of data hazard

$\overset{\text{decode , check for structural hazard}}{\text{Issue}}$

$$\underbrace{\text{Issue RDOp}}_{\text{ID}} \rightarrow \text{wait until no data hazards , then read operands}$$



Have WAR hazards:

DIVD F0
ADDD F10,F0,F8
SUBD F8

2. Techniques:

- Scoreboarding
- Tomasulo algorithm

Have WAW hazards:

DIVD F0
ADDD F8,F0
SUBD F8

Called: Dynamically scheduled or out-of-order execution machines

Scoreboarding

- Allows out of order execution, stalling if WAR,WAW
- Multiple instructions in the EX stage → multiple FU's
- Example: 2 MPYD, 1ADDD, 1DIVD, 1Integer (mem/br/ops)
- Scoreboard : Structure where a record of the data dependences is constructed (at issue stage)

  → controls: 1 when instr can RDOp

  2 when instr can execute

  3 when instr can write to reg

Inst Steps (No mem)

- Issue: if FU is free and no active inst has same dest register (WAW) → issue

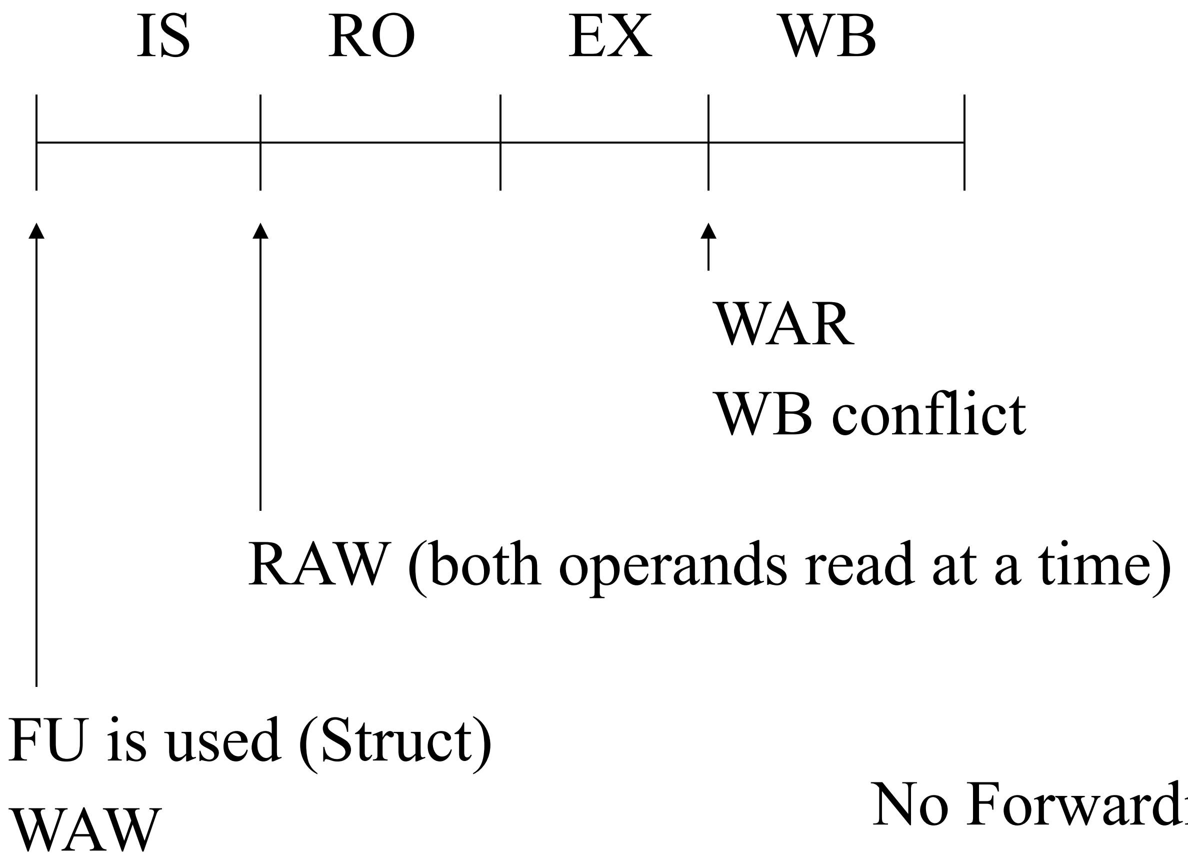
  → else stall [this inst and following ones]
- RDOp : wait until source ops are available (no one is in the process of writing) (RAW).

  Regs only read when both available
- EX:
- WB: Stall if WAR hazard

$$\left\{\begin{array}{l}\text{DIVD F0}\\ \text{ADDD F10,F0,F8}\\ \text{SUBD F8}\end{array}\right.$$

  or WB conflict
- No Forwading

Summary



No Forwarding

Note: FU free after EX

See Figure C.55

Parts of Scoreboard

- Instruction status : where the instructions are
- FU status : State of FU
- Reg status : Which FU will write it

See Figure C.56 and Figure C.57

What Limits Scoreboarding ?

- Amount of parallelism in instructions (better be beyond BB)
- # of Scoreboard entries (instruc. window)
- # and types of FU's → structural hazards
- presence of <u>WAR, WAW</u>

↓

can be removed with

register renaming → use "virtual registers"

↓

Tomasulo's algorithm

Tomasulo

- basic ideas
  - Reserv. stations fetch and buffer ops as soon as they are available $\longrightarrow$ no need to operate from registers
  - As instructions are issued : reg specifiers for pending operands are renamed to names of reserv. stations
    $\longrightarrow$ Register renaming that avoids WAW and WAR

Renaming

```
DIV     F0,F2,F4
ADD    F6,F0,F8
S         F6,0(R1)
SUB    F8,F10,F14
MUL    F6,F10,F8
```

```
DIV    F0,F2,F4
ADD    S,F0,F8
S         S,0(R1)
SUB    T,F10,F14
MUL    F6,F10,T
```

Tomasulo

- Register renaming provided by the reservation stations:
  - buffer the operands of instructions waiting to execute
  - Pending instructions designate the reservation station that will provide their input →effectively a register
  - When successive writes to a register overlap in execution, only the last one is actually used to update the register
- Other characteristics of Tomasulo:
  - Hazard detection and execution control are distributed
  - Bypassing everywhere (use the common data bus CDB – all units waiting for a result can loaded simultaneously)

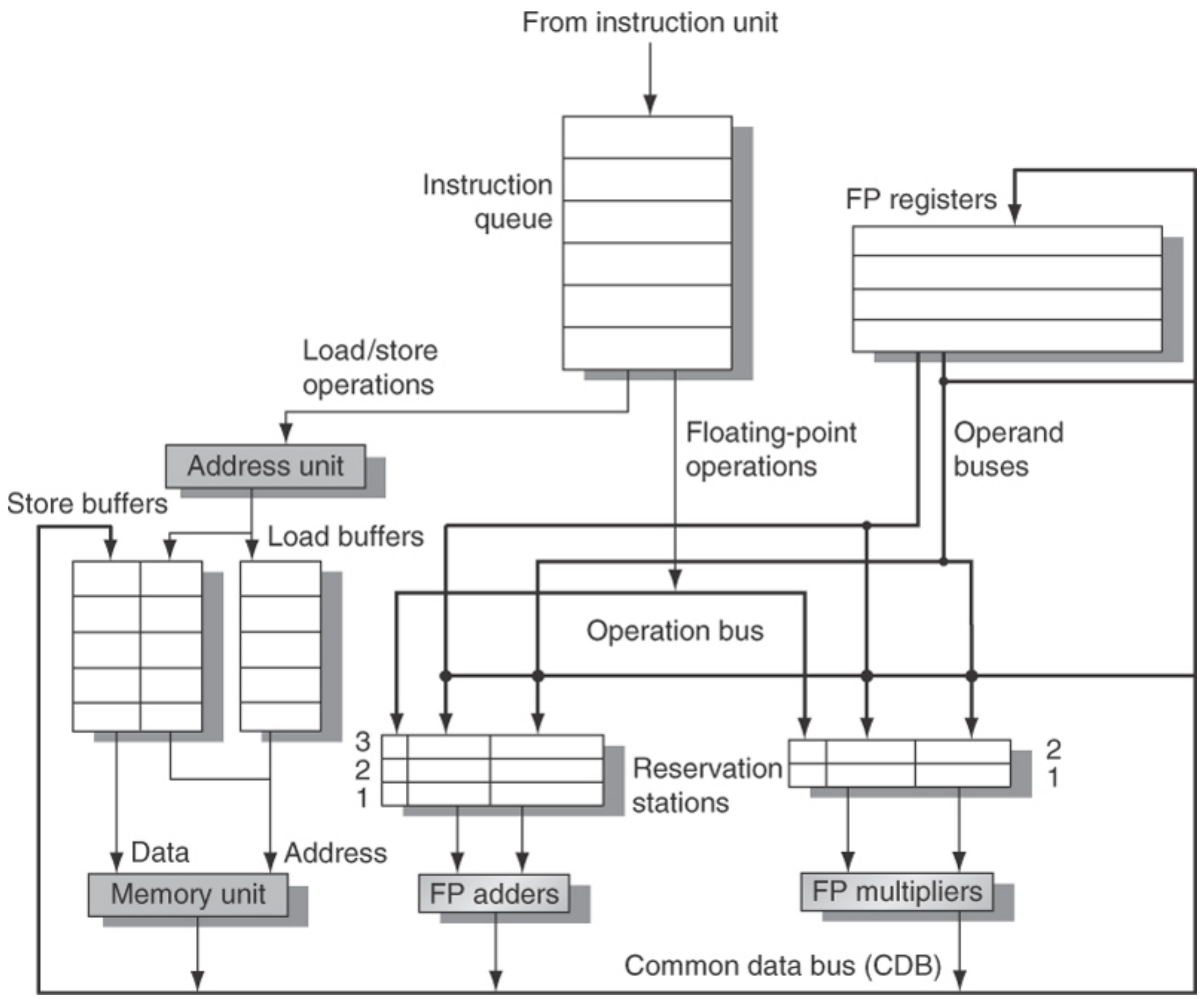

From instruction unit
Instruction queue
FP registers
Load/store operations
Floating-point operations
Operand buses
Address unit
Store buffers
Load buffers
Operation bus
3
2
1
Reservation stations
2
1
Data
Address
Memory unit
FP adders
FP multipliers
Common data bus (CDB)

Figure 3.6

Components

- reservation stations : instr waiting execution
- Ld buffers : hold data/addr coming from mem
- St buffers : hold data/addr going to mem
- All buffers & res stations have tags for hazard control

Steps of an Instruction

- Issue :
  - get next instr from instruction queue
  - issue it to empty reservation station
  - send operands to the rs; if ops not ready, write the rs that will produce them
  - if no reserv stations / buffers: structural Hz , stall
  - This step renames registers, eliminating WAR and WAW
- EX:
  - monitor bus for available operand
  - when available, put it in rs
  - when all ops ready, execute
  - By delaying until all ops are available, handle RAW
  - Independent functional units can begin executing in the same cycle
  - If two rs in the same FU become ready in the same cycle, one is chosen to execute

Steps of an Instruction (Cont)

- EX (cont):
  - Ld/st have a two-step execution
  - First step: compute effective address when base register is available and then eff. addr. is placed in the load or store buffer
  - Second step: actual mem access
  - Ld/st are maintained in program order through effective addr calculation
  - For now: do not allow EX of any instruction following a branch until the branch is resolved (later: allow EX, not allow WB)
- WR:
  - write result on bus. From there, it goes to regs & res. Stations/buffers
  - If store: write to memory

Detecting and Eliminating Hazards

- Done by tags attached to rs, regs, buffers
- They are names for extended set of virtual regs used in renaming
- Tag: field that encodes a name for the rs and load buffs
- Rs/buffs are like registers
- Once an instruction is waiting for an operand, it refers to the operand with the tag number of the rs/buff that will produce it
- Since there are more rs than architectural registers → WAW and WAR hazards are eliminated by renaming results with rs

State of a Reservation Station

- Op : operation to be performed
- Qj Qk : reservation station that will produce the source operands, or …
- Vj Vk : value of source operands. For loads, the Vk holds the offset field
- A: Holds information for the memory address calculation for a ld/st. Initially, the immediate field of the instr is stored. After the address calculation, the effective address is stored.
- Busy : this is busy

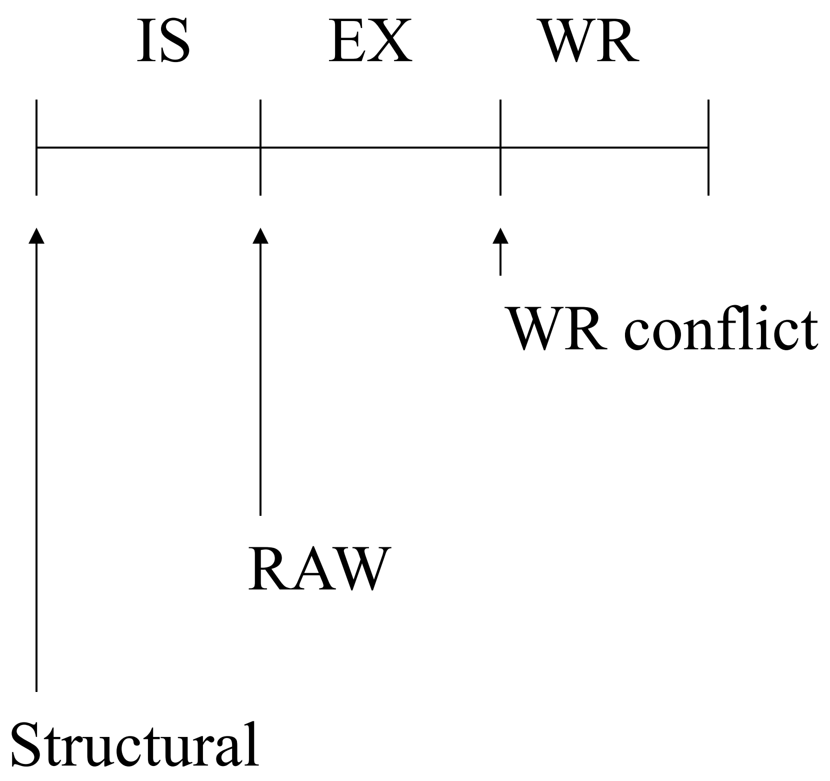
<u>Register file</u>

- Qi : rs that is computing a value to store here

<u>Load/Store buffs</u>

- A: effective address

Summary



Typical assumptions:

- IS,WR take one cycle each
- One instruction IS per cycle
- Functional Units (FUs) not pipelined
- Results are communicated via the CDB
- Assume you have as many load/store buffers as needed
- Loads/stores take 1 cycle to execute
- Loads/stores share a memory access unit
- Stores and branches do not have WR
- If an instruction is in its WR stage in cycle x, then an instruction that is waiting on the same FU (due to a structural hazard) can start executing on cycle X, unless it needs to read the CDB, in which case it can only start executing on cycle X+1
- Only one instruction can write to the CDB in a clock cycle
- Whenever there is a conflict for the FU, assume that the first (in program order) of the conflicting instructions gets access, while the others are stalled. This includes possible WR conflicts
- When an instruction is done executing in its functional unit and is waiting for the CDB, it is still occupying the functional unit and its reservation stations, and no other instruction may enter

See Figure 3.7

Differences over scoreboard

1. Value of operand in one of the fields of a rs is read from the output of FU, not from a reg
2. WAR: ADDD can complete before DIVD initiates

Advantages of Tomasulo

1. Distributed hazard detection logic: multiple instructions waiting on a single result:broadcast in CDB releases all
2. Removes stalls for WAW,WAR

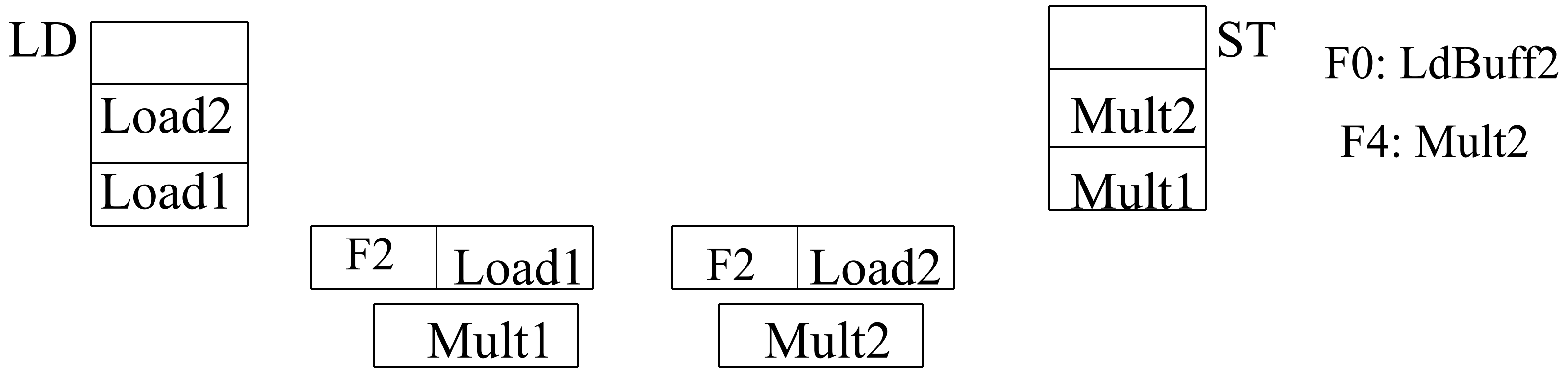
```
       { LD    F6,…..
WAR    { DIVD  F10 F0 F6  ; rs points to Load1 reservation station
WAW    { ADD   F6 F8 F2   ; reg file receives output of Add2, not of
                            Load1
```

See Figure 3.8

More on elimination of WAW, WAR

```
Loop :  LD        F0,0(R1)
        MULTD     F4,F0,F2
        SD        F4,0(R1)
        DADDUI    R1,R1,-8
        BNE       R1,R2,Loop
```

- predict that branches will be taken $\longrightarrow$ loop is unrolled dynamically by the hardware (no need many regs)



See Figure 3.10

…however , need dynamic disambiguation of address
    stall if addr Ld2 = addr pending stores (or forward)
    or if addr St2 = addr pending loads or stores

else , could execute iterations out of order

pbms: hardware intensive
    CDB bottleneck (if replicate, replicate logic too)

key features dynamic scheduling
    register renaming
    dynamic memory disambiguation