ECE 3057: Architecture, Concurrency and Energy in Computation Summer 2019

Lecture 7: Forwarding

David Anderson

School of Electrical and Computer Engineering
Georgia Institute of Technology

Acknowledgment: Lecture slides adapted from GT ECE 3056 (t. Krishna & S. Yalamanchilli), MIT 6.823 (Arvind and J. Emer) and CMU 18-447 (O. Mutlu)

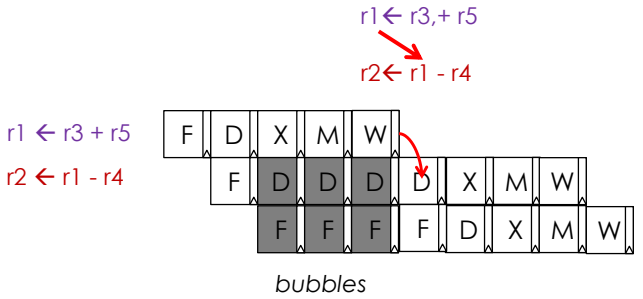
Announcements

- Syllabus and Schedule
  - Pipelining will be completed this Thursday
  - Quiz 1 is Tuesday, June 11
    - Topics: single-cycle, multi-cycle, pipelining, hazards, CPI, forwarding
  - Lab Assignment 1 due next Thursday, June 6
    - Doing the lab will be extremely useful for understanding concepts for Quiz 1
- Problem Sets
  - Problem set on Single Cycle, Multi Cycle, and pipelining added today

Resolving Data/Control Hazards

- Strategy 0: remove dependence
  - Detect and eliminate the dependence at the software level
- Strategy 1: stall
  - *Wait for the result to be available by freezing earlier pipeline stages*
- Strategy 2: bypass/forward
  - *Route data as soon as possible after it is calculated to the earlier pipeline stage*
- Strategy 3: speculate
  - *Guess the result and proceed*
    - ***Guessed correctly*** → *no special action required*
    - ***Guessed incorrectly*** → *kill and restart*
- Strategy 4: do something else
  - *Switch to some other task/thread while dependence resolves*

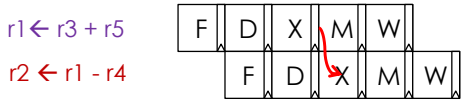
Forwarding: Key Idea



Instruction interlocked/stalled in decode stage

But when is data actually available?

At Execute



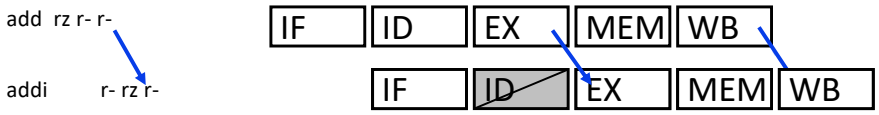
Forward to ALU with no bubbles

Data Forwarding (or Data Bypassing)

- It is intuitive to think of RF as state
  - "add rx ry rz" literally means get values from RF[ry] and RF[rz] respectively and put result in RF[rx]
- But, RF is just a part of a communication abstraction
  - "add rx ry rz" means

    1. get the results of the last instructions to define the values of RF[ry] and RF[rz], respectively,

    2. until another instruction redefines RF[rx], younger instructions that refer to RF[rx] should use this instruction's result
- What matters is to maintain the correct "data flow" between operations, thus



Implementing a Forward

- Detect the dependence
  - Instructions I_A and I_B (where I_A comes before I_B) have RAW dependence iff
    - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
    - dist(I_A, I_B) ≤ dist(ID, WB) = 3
  - How do we detect this?
    - Same as before, as we did for stall!
- Forward
  - if I_B in ID stage reads a register written by I_A in EX, MEM or WB stage, then the operand required by I_B is not yet in RF
    - retrieve operand from datapath instead of the RF
    - retrieve operand from the youngest definition if multiple definitions are outstanding
  - How?

Example - Stall

I1: SUB $t5, $t0, $t0
I2: XOR $t6, $t5, $t1
I3: SW $t6, 0($t2)
I4: LW $t1, 0($t2)
I5: ADD $t3, $t1, $t0

CPI? 14/5 = 2.8

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
SUB	F	D	E	M	W															
XOR		F	D	D	D	D	E	M	W											
SW			F	F	F	F	D	D	D	D	E	M	W							
LW							F	F	F	F	D	E	M	W						
ADD											F	D	D	D	D	E	M	W		

Example - Forwarding

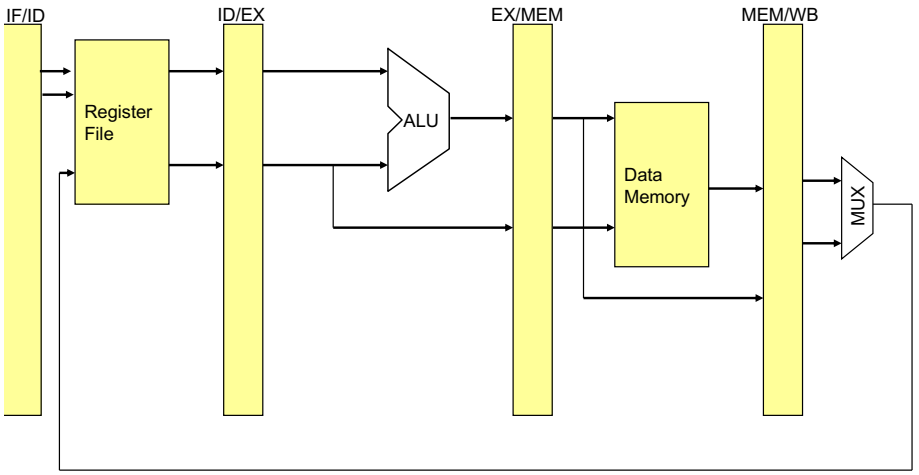
I1: SUB $t5, $t0, $t0
I2: XOR $t6, $t5, $t1
I3: SW $t6, 0($t2)
I4: LW $t1, 0($t2)
I5: ADD $t3, $t1, $t0

CPI? 6/5 = 1.2

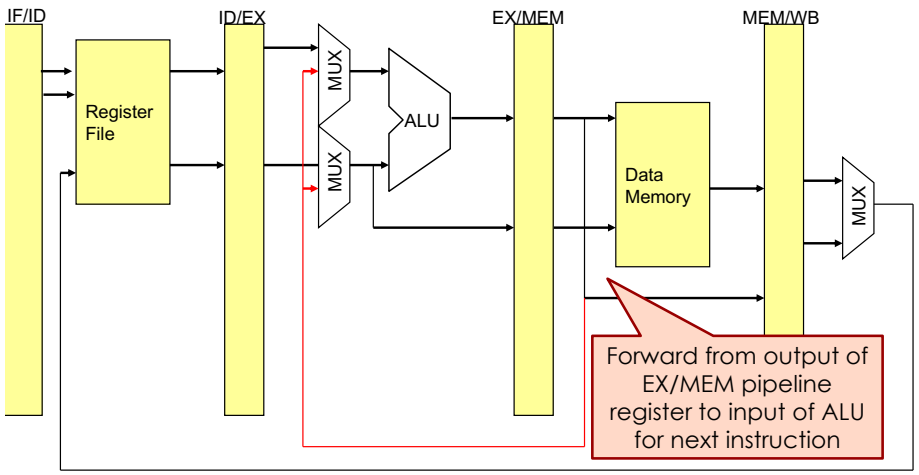
Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
SUB	F	D	E	M	W															
XOR		F	D	E	M	W														
SW			F	D	E	M	W													
LW				F	D	E	M	W												
ADD					F	D	D	E	M	W										

An ALU instruction in Decode has to stall for a Load instruction in Execute

Implementing Forwarding



Forwarding (from EX/MEM)



Example - Forwarding

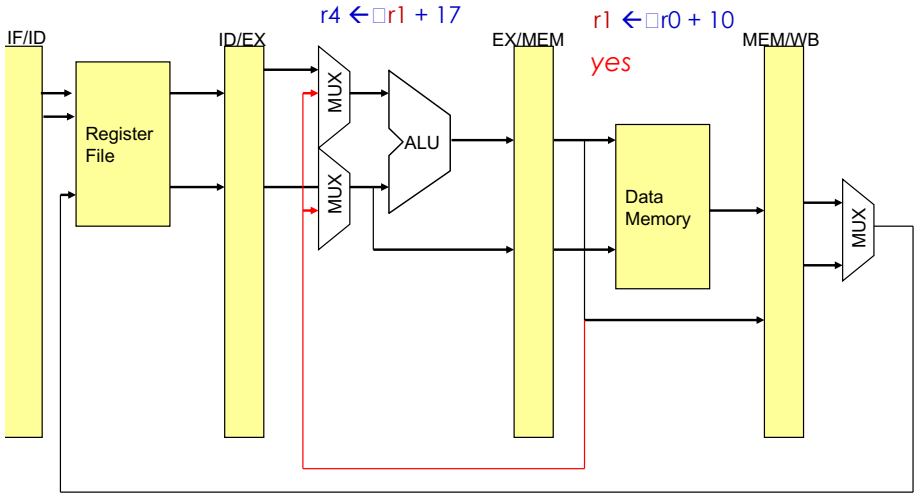
I1: SUB $t5, $t0, $t0
I2: SW $t6, 0($t2)
I3: XOR $t6, $t5, $t1
I4: LW $t1, 0($t2)
I5: ADD $t3, $t6, $t0

CPI? 5/5 = 1

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
SUB	F	D	E	M	W															
SW		F	D	E	M	W														
XOR			F	D	E	M	W													
LW				F	D	E	M	W												
ADD					F	D	E	M	W											

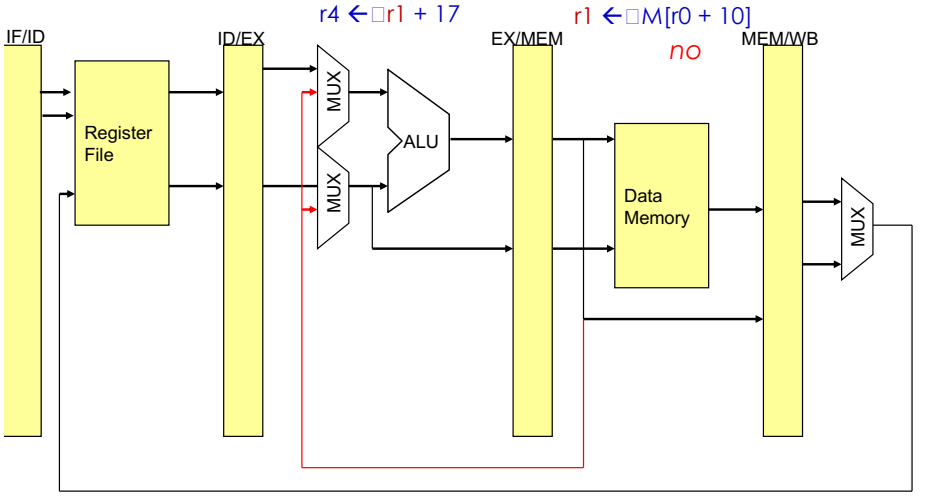
Forwarding (from EX/MEM)

Does this bypass help?

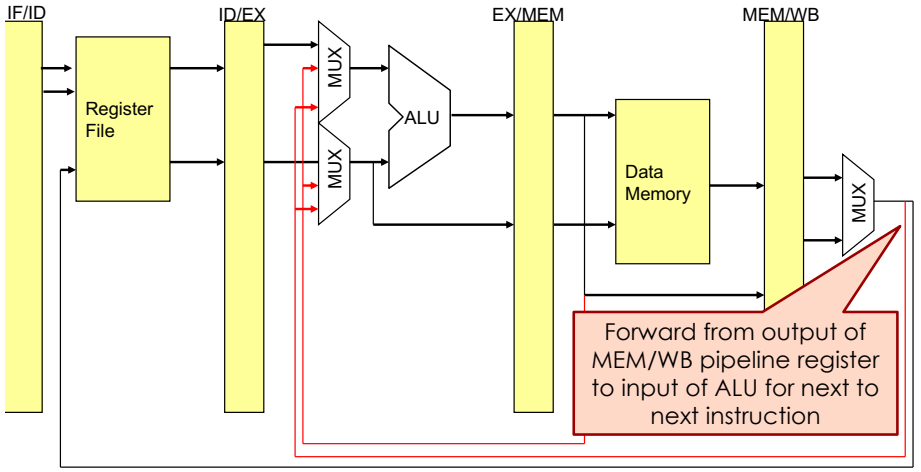


Forwarding (from EX/MEM)

Does this bypass help?

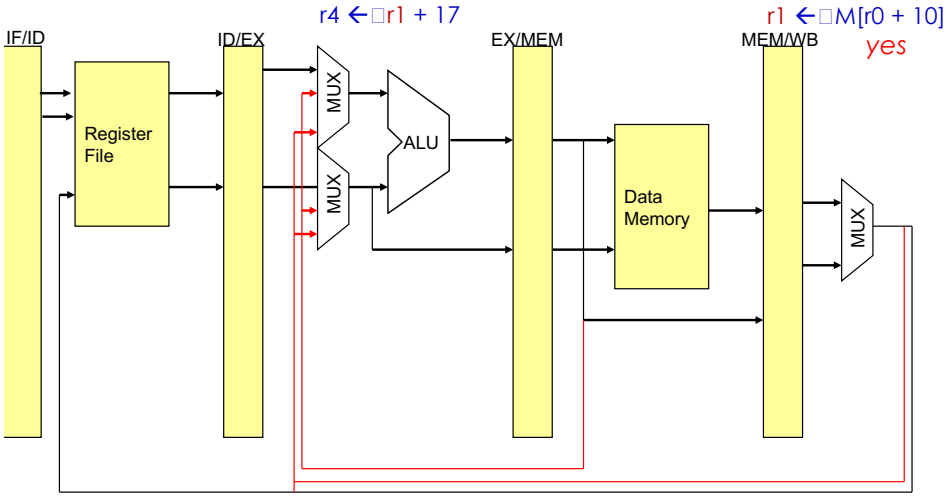


Forwarding (from MEM/WB)



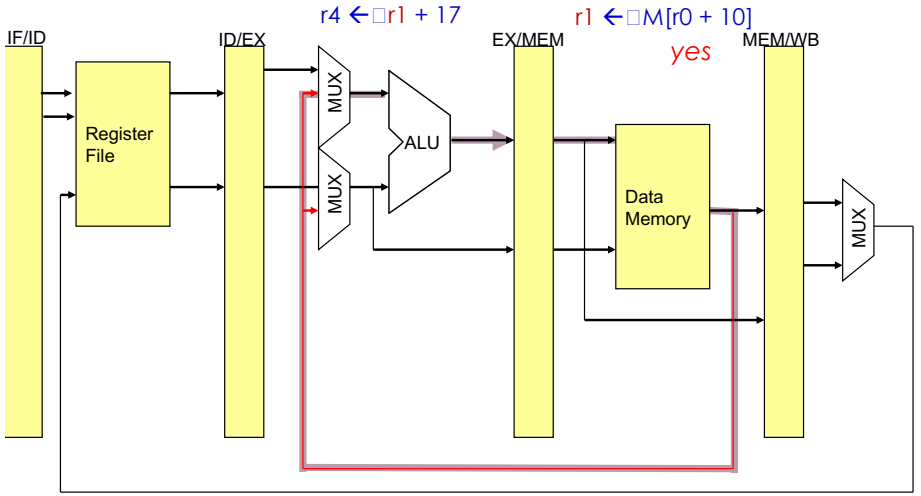
Forwarding (from MEM/WB)

Does this bypass help?



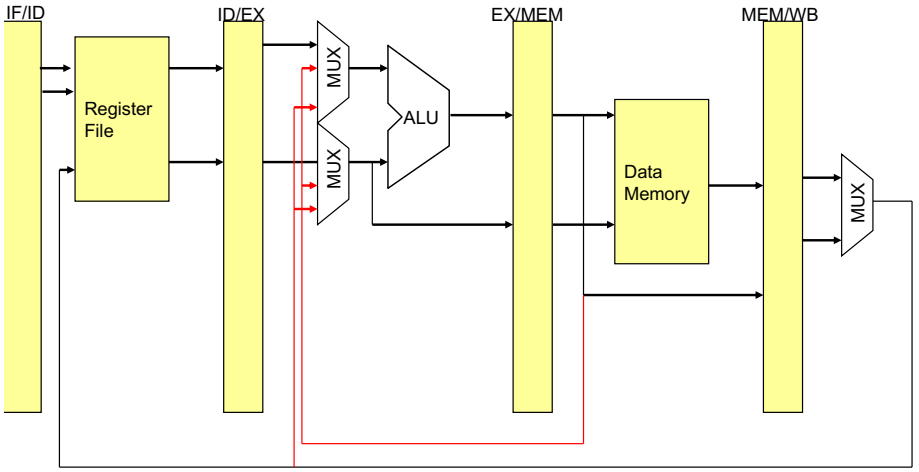
Forwarding (from MEM/WB)

Does this bypass help?

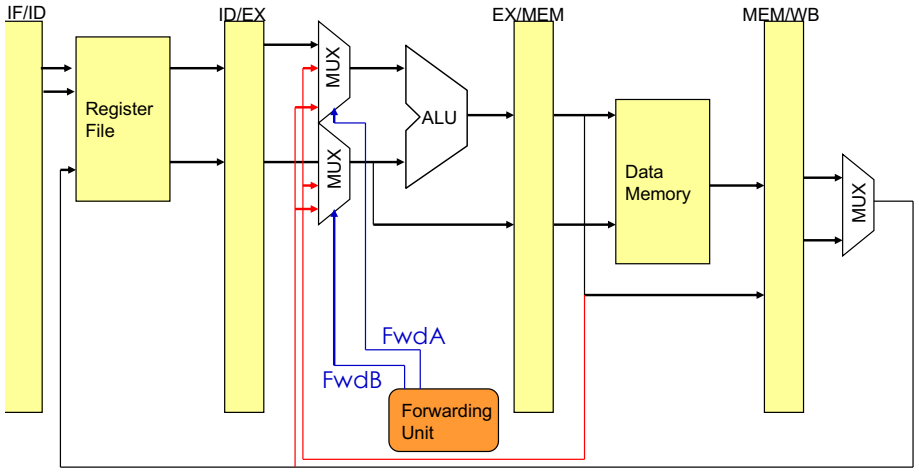


But is this a reasonable design? No. Critical path: (MEM + EX)

Forwarding (from EX/MEM & MEM/WB)



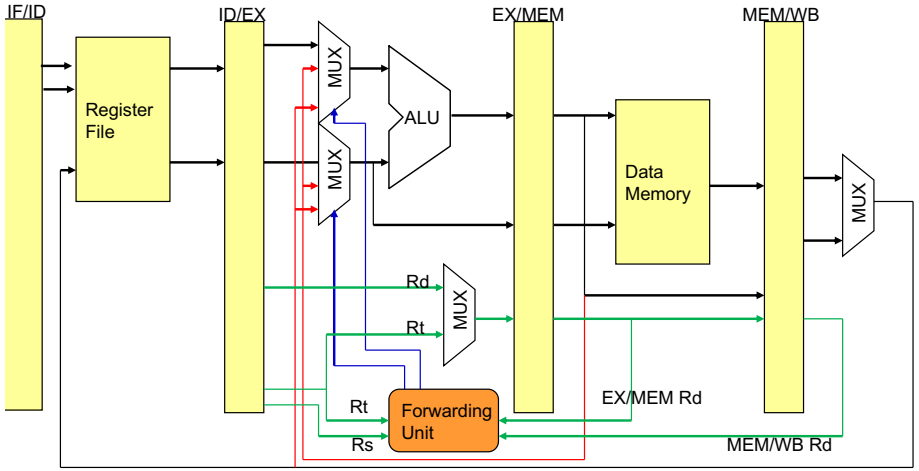
Forwarding (operand selection)



Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    FwdA → EX/MEM
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    FwdB → EX/MEM
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    Fwd A→ MEM/WB
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    Fwd B → MEM/WB

Forwarding (operand propagation)



Combinational Logic!

Implementing a forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
  (EX/MEM.RegisterRd == ID/EX.RegisterRs or
  EX/MEM.RegisterRd == ID/EX.RegisterRt)
  and EX/MEM.RegWrite
  and EX/MEM.RegisterRd ≠ 0

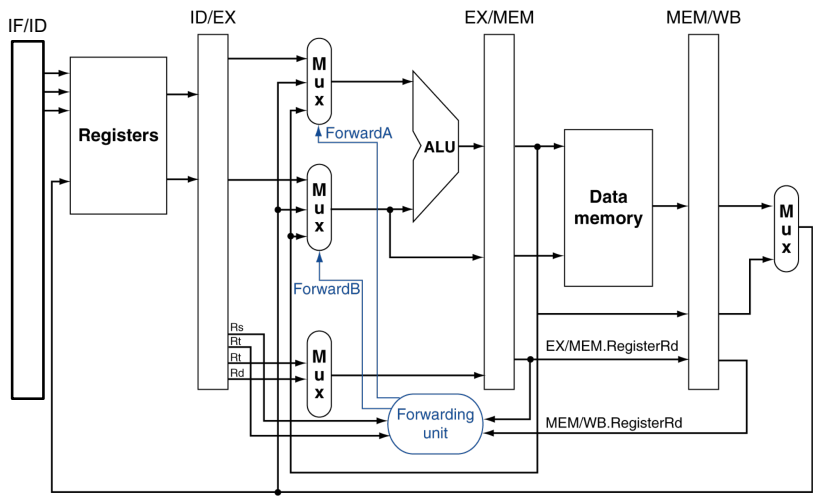
  Fwd from EX/MEM pipeline reg

  OR

  (MEM/WB.RegisterRd == ID/EX.RegisterRs or
  MEM/WB.RegisterRd == ID/EX.RegisterRt)
  and MEM/WB.RegWrite
  and MEM/WB.RegisterRd ≠ 0

  Fwd from MEM/WB pipeline reg

Datapath with Forwards



Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

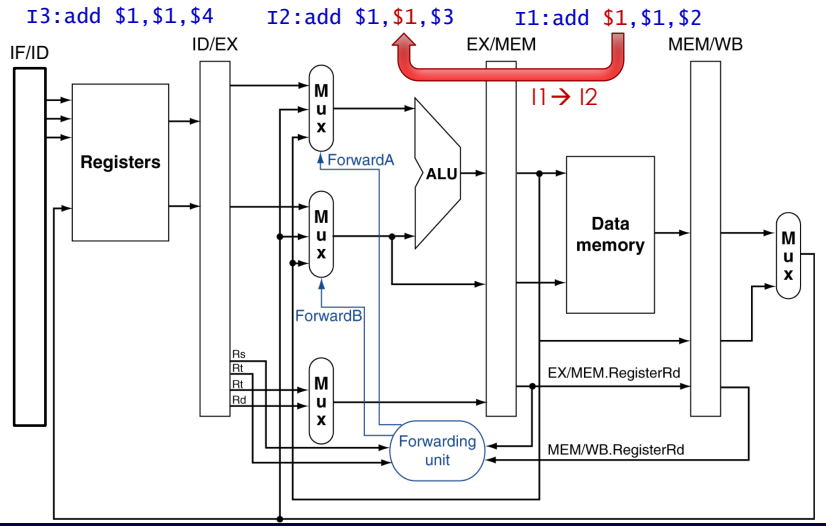
What if both EX and MEM hazards?

- Consider the sequence:

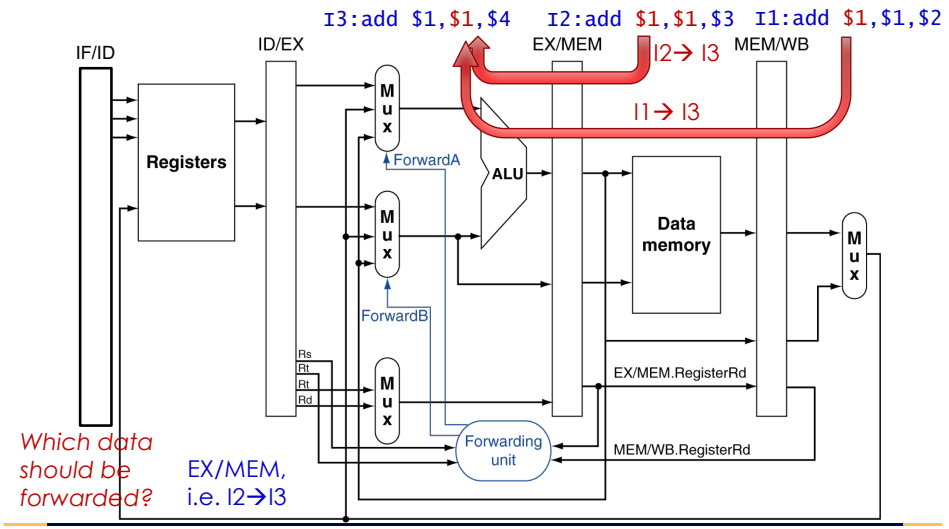
```
I1: add $1,$1,$2
I2: add $1,$1,$3
I3: add $1,$1,$4
```

- Which data should be forwarded to I2?
  - I1
- Which data should be forwarded to I3?
  - I2

What if both EX and MEM hazards?



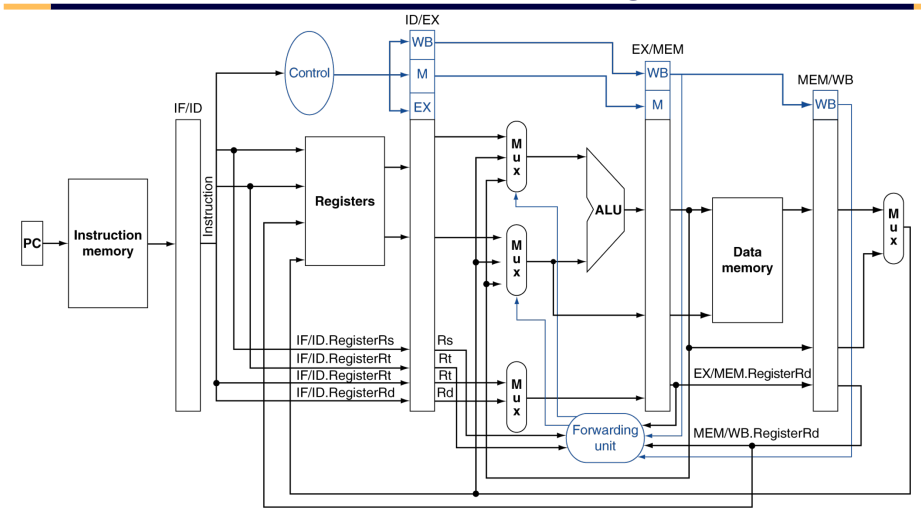
What if both EX and MEM hazards?



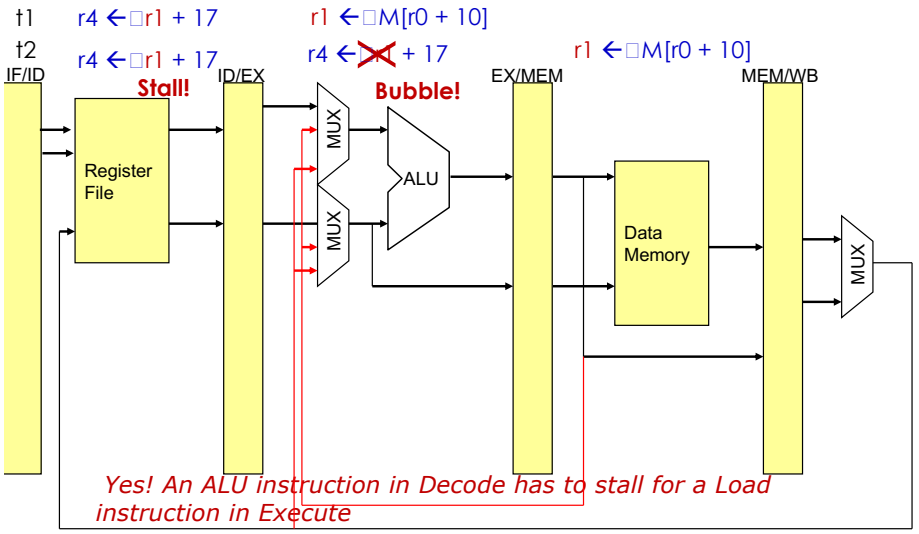
Revise MEM hazard forwarding condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Datapath with forwarding



Is there still a need for a stall signal?



Summary of Forwarding

- Bypassing helps reduce stalls by forwarding data from EX/MEM or MEM/WB pipeline registers to the ALU
- Full bypassing may be too expensive to implement
  - typically all frequently used paths are provided
  - some infrequently used bypass paths may increase cycle time and counteract the benefit of reducing CPI
- Loads produce stalls
  - Instruction after load cannot use load result in next cycle
  - MIPS-I ISA defined *load delay slots*, a software-visible pipeline hazard (compiler schedules independent instruction or inserts NOP to avoid hazard). Removed in MIPS-II.
- Jumps and Branches may cause bubbles
  - next