CS184b: Computer Architecture (Abstractions and Optimizations)

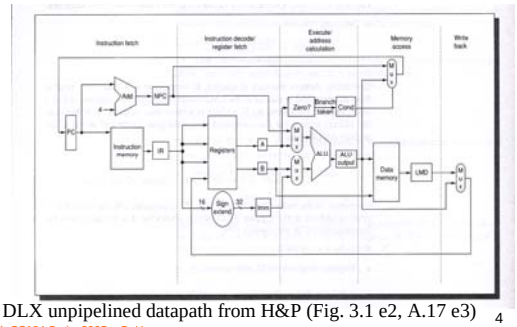
Day 3: April 1, 2005
Pipelined ISA

Today

- Pipelined Processor Issue
  - Hazards
    - structural
    - data
    - control
  - accommodating
  - Impact

Pipelining ISA

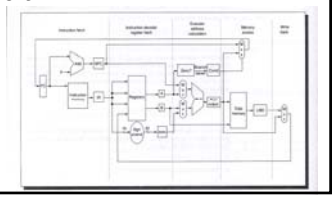
DLX Datapath



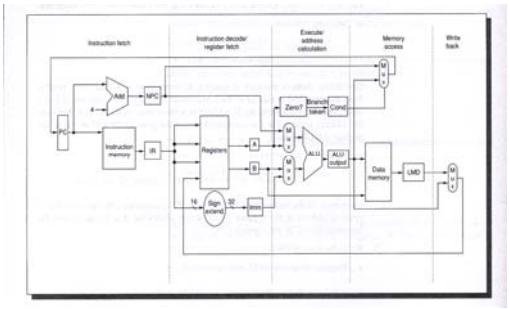
DLX unpipelined datapath from H&P (Fig. 3.1 e2, A.17 e3)

DLX Model Behavior

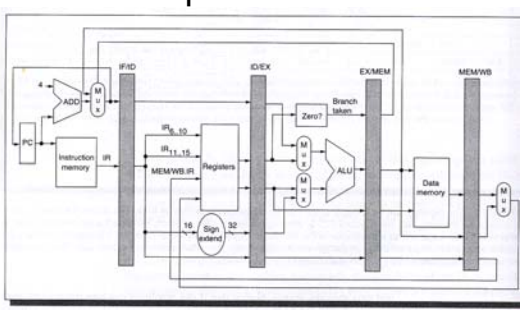
- Lookup Instruction
- Read Registers
- Perform primitive ALU Op
- Read/Write memory
- Write register result



Pipeline?



Pipeline DLX



DLX pipelined datapath from H&P (Fig. 3.4e2, A.18 e3)

Hazards

- Structural (resource bound)
- Data (value timing)
- Control (knowing where to go next)

Structural Hazards

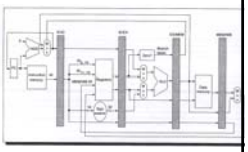
- Arise when
  - Instructions have varying resource requirements
    - E.g. write port to RF is a resource
  - Usage of resources not occur at same time
  - Not want to provide resources for worst-case
    - typically because it's not the "common" case
    - performance impact small compared to cost of handling worst case

Structural Hazards

- Have to consider:
  - all possible overlaps of instructions
  - simplified by considering instruction classes
    - (e.g. add R1,R2,R3, sub R3,R4,R5, … all use same resource set…)

Structural Hazard: Identify

- Identify by:
  - looking at instruction (class) resource usage in pipeline
- *E.g.* Register-Register Op:
  - IF - I-mem port
  - ID - 2 register read ports
  - EXU - ALU
  - MEM - ---
  - WB - 1 register write port



Structural Hazard: Identify

- R-R: 1I 2RR 1A -- 1RW
- L/S: 1I 1RR 1A 1AB,1DB 1RW
- BR: 1I 1RR 1A -- --

- Conflicts:
  - standard DLX
  - RF has 1R, 1RW port

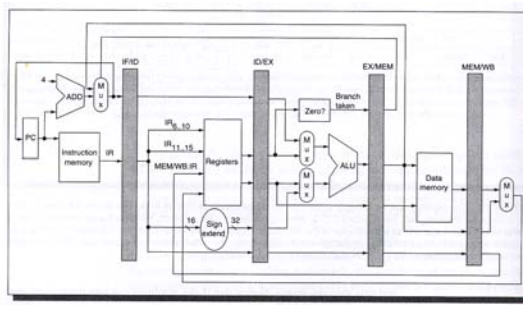
Structural Hazard: Identify

- Pipelined Memory access
- R-R: 1I 2RR 1A -- -- 1RW
- L: 1I 1RR 1A 1AB 1DB 1RW
- S: 1I 1RR 1A 1AB,1DB -- 1RW
- BR: 1I 1RR 1A -- --

Structural Hazards: Deal

- The datapath cannot handle
- Always have the option of idling on a cycle
  - "Bubble" into pipeline
  - allow downstream continue, stall upstream
- Options:
  - detect when occurs and stall one instruction
  - detect will occur at issue and stall

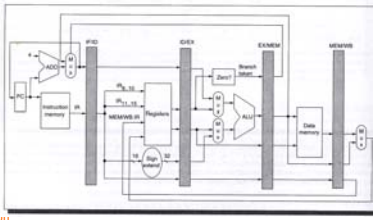
Pipeline DLX



Data Hazards

Data Hazards

- Pipeline Latency
- Instruction effects not completed before next operation begins

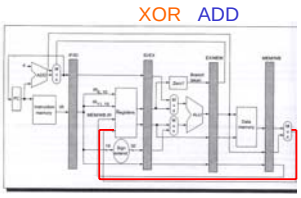


Data Hazard: Example

- ADD R1,R2,R3
- XOR R4,R1,R5

Data Hazard: Example

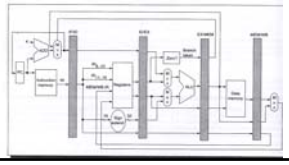
- ADD R1,R2,R3
- XOR R4,R1,R5



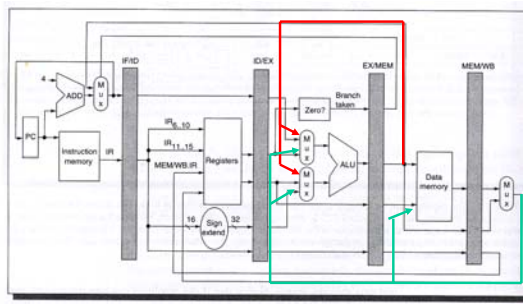

```
IF  ID  EX  MEM WB
    IF  ID  EX  MEM WB
        IF  ID  EX  MEM WB
            IF  ID  EX  MEM WB
                IF  ID  EX  MEM WB
```


Data Hazard: Solving

- Primary problem (DLX)
  - data not back in RF
  - read stale data
- Partially solve with bypassing
  - when good data exist before use



Data Hazard: Solving

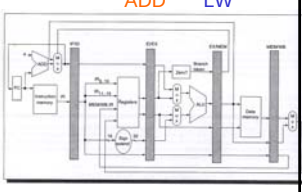


Data Hazard

- Note: since ops may stall, interrupt, resume
  - cannot decide how to set bypass muxes in ID stages
  - have to determine based on state of pipeline

Data Hazard

- Not all cases can bypass
  - if data not available anywhere, yet...
  - *e.g.*
    - LW R1,4(R2)
    - ADD R3,R1,R4




```
IF  ID  EX  MEM WB
    IF  ID  EX  MEM WB
        IF  ID  EX  MEM WB
            IF  ID  EX  MEM WB
                IF  ID  EX  MEM WB
```


Model/Common Case

- Optimize for Common/simple case
- Implementation transparency
  - Hide fact/details of pipelining
- Could have slowed the initiation interval for all ops
- **OR** could have said can never use value for number of cycles
- But, only few sequences/cases problematic
  - let rest run faster

Types of Data Hazards

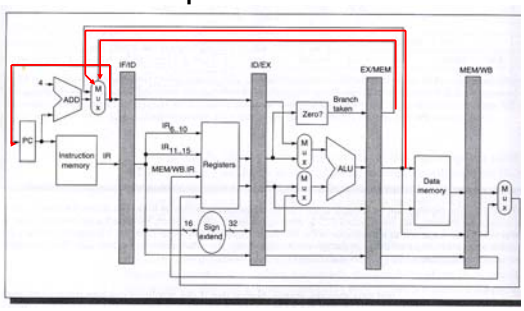
- RAW (example seen)
- WAW
  - order of writes transposed going to memory
  - leave wrong value in memory
- WAR
  - read gets value computed "after" it should have completed

Compiler

- Instruction Scheduling can try to avoid/minimize stalls
  - (making some assumptions about implementation)
- Schedule instructions in hazard slot
  - possible when have parallelism/independent tasks
- example where optimize across larger block give tighter results

Control Hazards

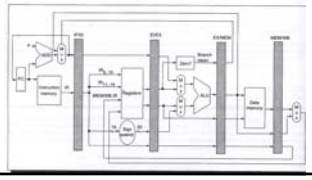
Pipeline DLX



DLX pipelined datapath from H&P (Fig. 3.4)

Control Hazard

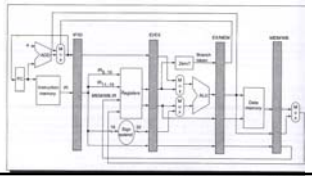
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB



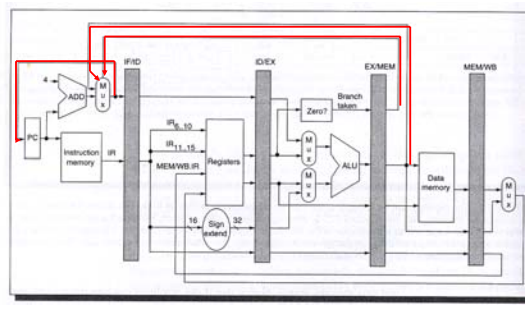
Control Hazard

IF ID EX MEM WB

IF ID EX MEM WB



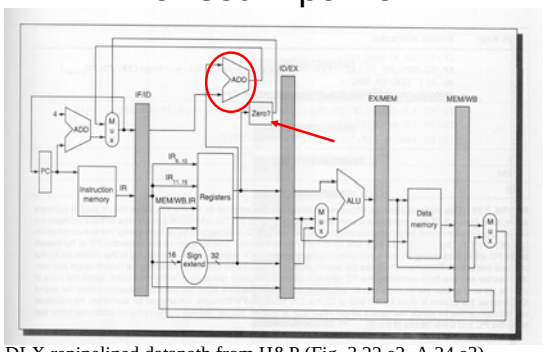
What can we do?



What can we do?

- Move earlier
  - tighten cycle
- "Guess" direction
  - predict (and squash)
- Accepted delayed transfer as part of arch.
  - Branch delay slot

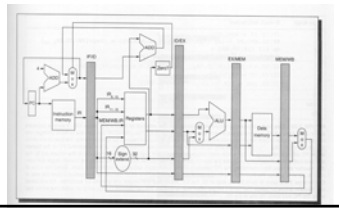
Revised Pipeline



DLX repipelined datapath from H&P (Fig. 3.22 e2, A.24 e3)

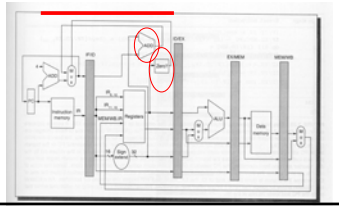
Consequence?

```
IF  ID  EX   MEM  WB
    IF  ID   EX   MEM  WB
        IF   ID   EX   MEM  WB
             IF   ID   EX   MEM  WB
                  IF   ID   EX   MEM  WB
```



Consequence

- Smaller cycle
- Longer ID stage delay
- Need separate Adder
- Not branch to reg.?

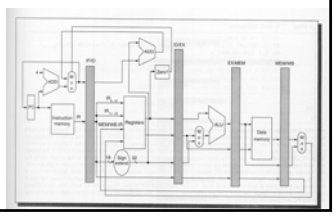


Pipeline

```
IF  ID  EX   MEM  WB
        IF   ID   EX   MEM  WB
             IF   ID   EX   MEM  WB
                  IF   ID   EX   MEM  WB
```

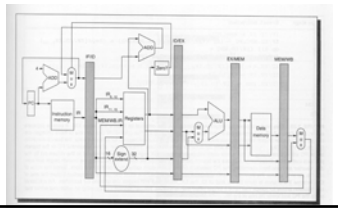
Avoiding Lost Cycles

- Do know where to go in not-taken case
  - just keep incrementing PC
- "Guess" not taken
- Begin Executing
- Squash if Wrong



Predict Branch Not Taken

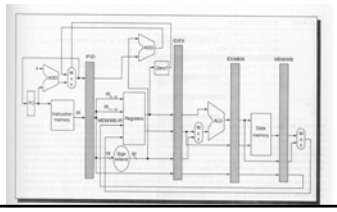
```
Branch:   IF  ID  EX   MEM  WB
Branch+1:     IF  ID   EX   MEM  WB
Branch+2:         IF   ID   EX   MEM  WB
                   IF   ID   EX   MEM  WB
```



Predict Branch not Taken (but **is**)

```
Branch:   IF  ID  EX   MEM  WB
Branch+1:     IF  ID   --   --   --
Target :          IF   ID   EX   MEM  WB
                   IF   ID   EX   MEM  WB
```

Squash ok:
no state change,
no effect of exec op



Avoiding Lost Cycle (2)

- Solve like load latency
  - separate load request
  - from load use
- Separate branch instruction (computation)
- From branch effect
- Architecturally specify
  - branch not take effect until X cycles later

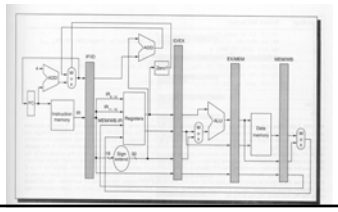
Branch Delay Slot

- SUB R1,R2,R3
- BEQZ R1,exit
- ADD R4,R5,R6 // always executed
- SUB R1,R4,R3

- exit:
- SW R3,4(R11)

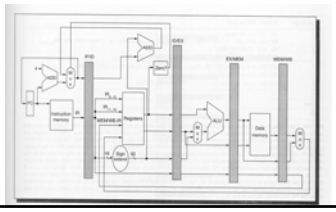
Branch Taken

```
Branch:   IF  ID  EX   MEM  WB
B-Delay:      IF  ID   EX   MEM  WB
Target :          IF   ID   EX   MEM  WB
                   IF   ID   EX   MEM  WB
```



Branch Not Taken

```
Branch:   IF   ID   EX   MEM  WB
B-Delay:       IF   ID   EX   MEM  WB
Branch+2:           IF   ID   EX   MEM  WB
                    IF   ID   EX   MEM  WB
```



More Control Fun to Come...

- Knowing what to run next can be big limit to exploiting parallelism (deep pipelining)
- ILP need more branch prediction

Exceptions

(cover if time available)

Skip to Big Idea Wrapup

What?

- Control transfer away from "normal" execution
- Every instruction
  - a conditional, multiway branch?
  - (branch and link)

What (examples)

- Page Fault
- System call
- Interrupt (event)
  - io
  - timer
- Unknown Instruction

Why?

- Cases represented are uncommon
- Instructions explicitly checking for cases add cycles
  - …lots of cycles to check all cases
  - when seldom occur
- Long instructions to describe all ways can branch
  - more compact to "configure" implicit places to go

Properties/Types

- synch/Asynch
- request/coerced
- maskable?
- within/between instructions
- resume/terminate

How make implementation difficult?

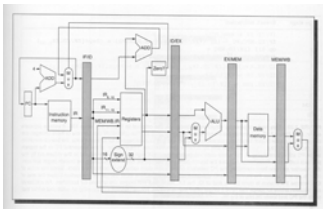
- Need to preserve sequential instruction abstraction
- Creates cause to see what happens between instructions
  - need to provide clean state view
- Instruction fail and need to be "restarted"
  - e.g. page fault

Hard cases

- Synchronous
- within instruction
- restartable

- latencies or parallelism allow out-of-order completion

Hazards



- LW R1,4(R2)
- ADD R3,R4,R3
- Case:
  - DLX
  - Pipeline where WB can occur before MEM
- May be correct to complete ADD
  - no hazards
  - but not restartable when fault on LW address

Restart Hazards

- LW R1,4(R2)
- ADD R3,R4,R3

- Restart/rerun
  - can get wrong answer by executing instruction again

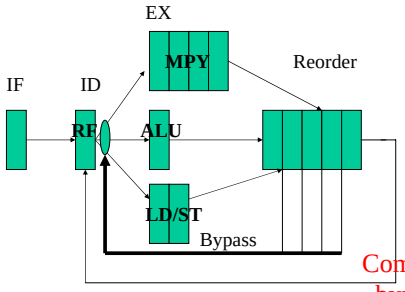
Solutions

Theme: save state

Re-Order Buffer

- Continue to execute
- Write-back to register file in-order
- Buffer results between completion and WB
- Bypass with newer results

Re-Order

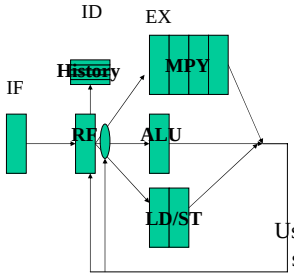


Complex (big) bypass logic.

History Buffer

- Keep track of values overwritten in register file
- Can restore old state from there

History



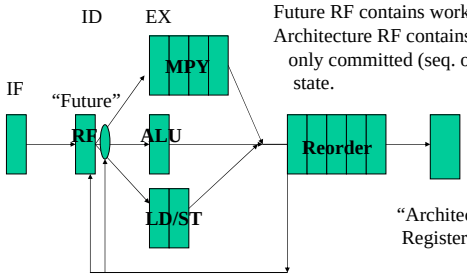
History Buffer contain:
PC Reg. # prev. reg value

Use history to "rollback" state of computation to consistent/committed point.

Future File

- Keep two copies of register file
  - committed / visible set
  - working set

Future



Future RF contains working state
Architecture RF contains only committed (seq. order) state.

"Architecture" Register File

Memory

- Note: may need to do re-order/bypass to memory as well
  - same issue as RF
  - not want to make visible state change
  - may want to run ahead (avoid adding dep.)
- Bigger issue as we go to longer latencies, OO-issue, etc.

Big Ideas [MSB]

- Preserve the (simple, stable) model
- While providing high-performance implementation

Big Ideas [MSB-1]

- Pipelining – simplest form of parallelism
  - Non-pipeline underutilizes resources
- Ops with different requirements
  - Some cases can run faster than others
  - Fast in simple, common cases
  - Correct in others

Big Ideas [MSB-1]

- Challenge of deciding what to do next
  - cyclic dependency
- Minimizing cost thereof
  - pipeline structure (minimize latency)
  - branch delay
  - prediction
- Common Case
  - predictable
  - exceptions