CS 270: Computer Organization

Pipelined Implementation I

Instructor:

Professor Stephen P. Carl

Sewanee: The University of the South

Real-World Pipelines: Car Washes

Sequential

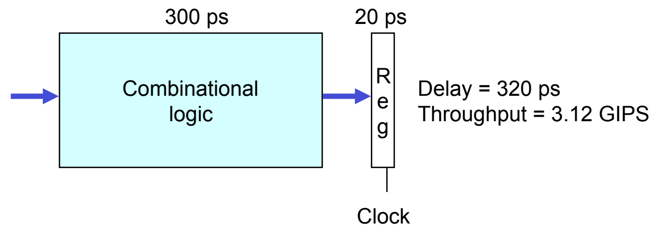
Parallel

Pipelined

The Basic Idea:

- Divide process into independent stages
- Move objects through stages in sequence
- At any given time, multiple objects are being processed

Computational Example

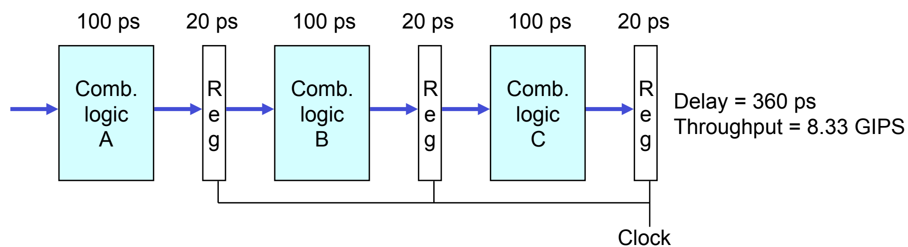


Sequential Design

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Requires a clock cycle of at least 320 ps
- GIPS = billions of instructions per second (1/delay)

Sewanee: The University of the South

3-Way Pipelined Version



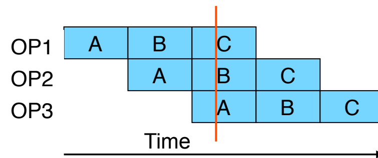
Pipelined Design

- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous one passes through stage A.
  - Begin new operation every 120 ps
- Increase overall latency, but also increase throughput!
  - 360 ps from start to finish, 8.33 billions of instructions/sec

Pipeline Diagrams

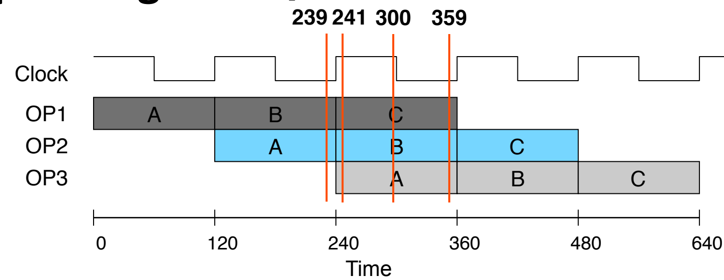
Unpipelined – cannot start new op until previous completes

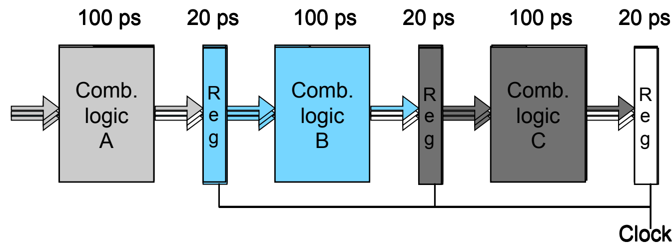
3-Way Pipelined – up to 3 ops in process simultaneously



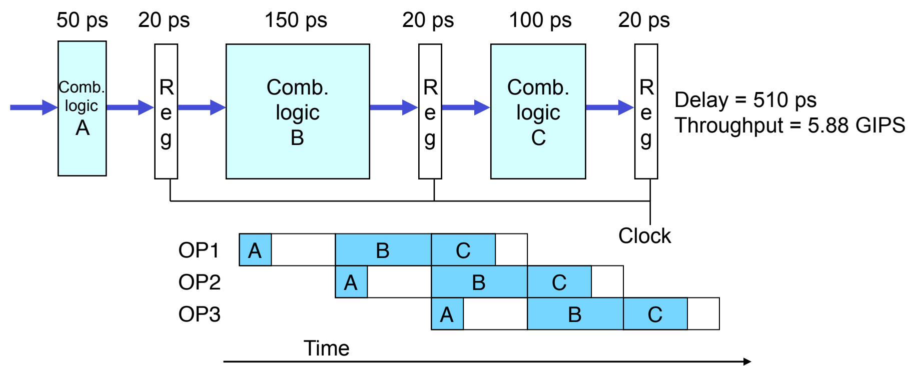
Sewanee: The University of the South

Operating the Pipeline



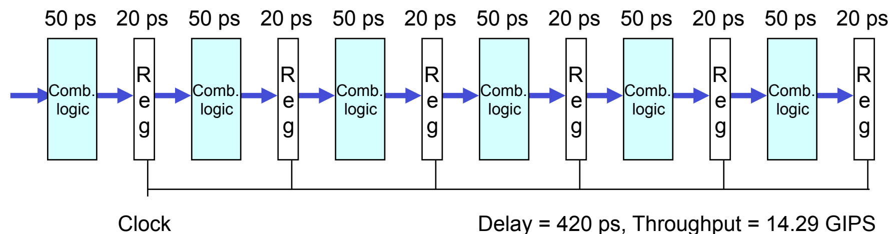


Limitations: Nonuniform Delays



- Throughput limited by slowest stage
- Other stages sit idle much of the time
- Challenge is to partition system into balanced stages

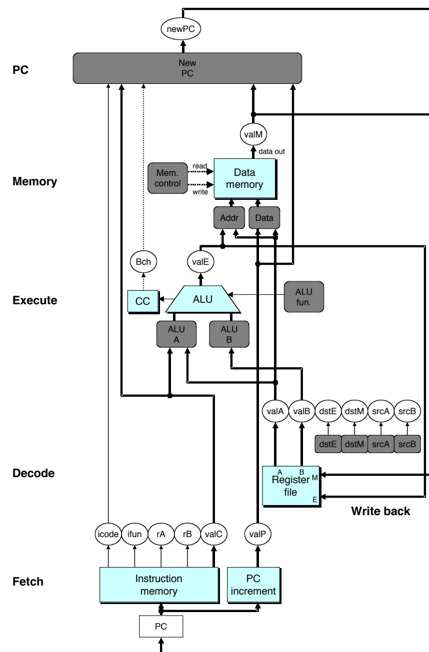
Limitations: Register Overhead



- As designers try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through *very* deep pipelining

SEQ Hardware

- Stages occur in sequence
- One operation in process at a time



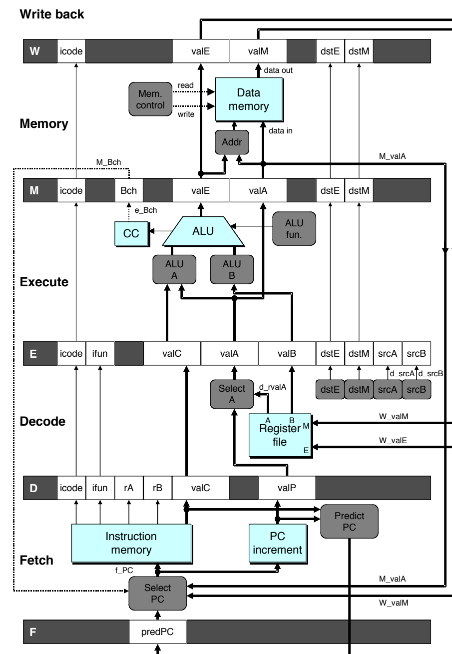
Sewanee: The University of the South

PIPE- Hardware

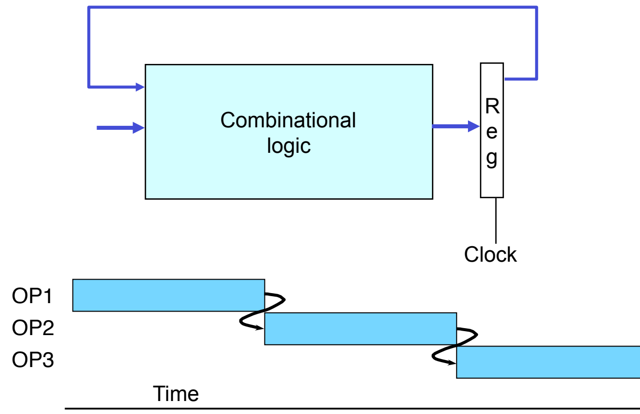
Pipeline registers hold intermediate values from instruction execution

Forward Paths (up-arrow)

- Values passed from one stage to next
- Cannot jump past stages (e.g., valC passes through decode)



Data Dependencies



Sequential System:

Each operation depends on result from preceding one

Data Dependencies in Y86 Code

```
1   irmovl $50, %eax
2   addl %eax , %ebx
3   mrmovl 100( %ebx ), %edx
```

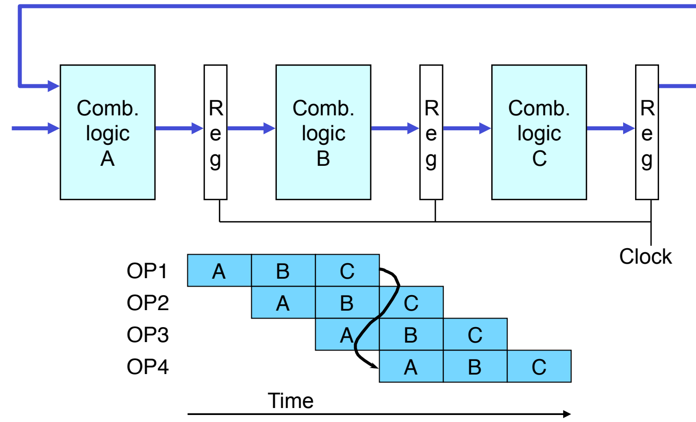
The result from one instruction is used as operand for another

- Read-after-write (RAW) dependency

Such dependencies are very common in actual programs;
our pipeline must handle these properly

- Ensure correct results, while minimizing performance impact

Data Hazards

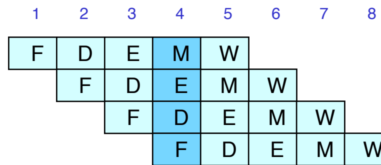


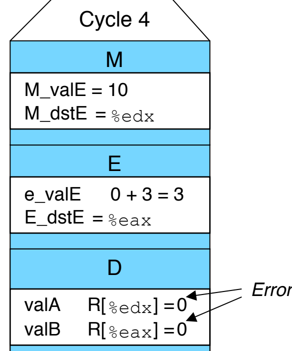
- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system!

Sewanee: The University of the South

Data Hazard: Problem

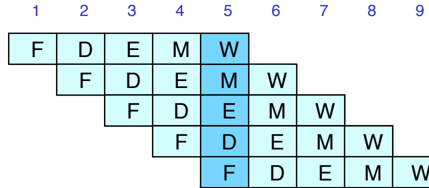
```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

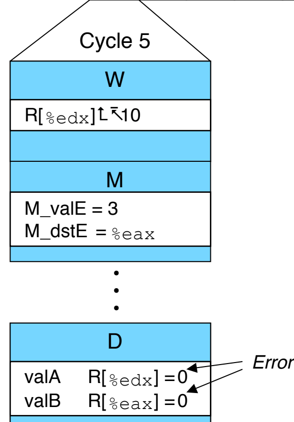




Fix? Try Inserting 1 NOP

```
# demo-h1.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: addl  %edx,%eax
0x00f: halt
```

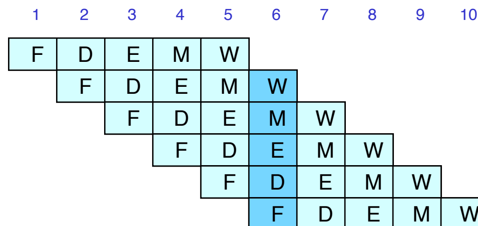


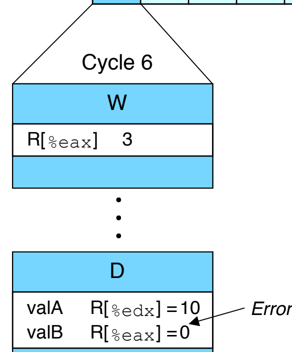


Sewanee: The University of the South

Data Hazard: Insert 2 NOP

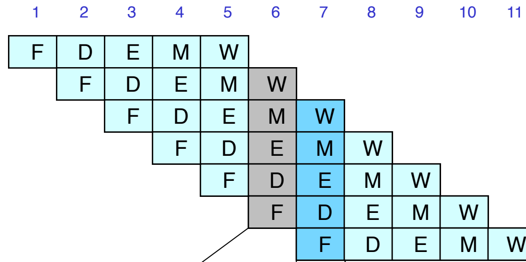
```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl  %edx,%eax
0x010: halt
```

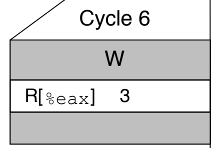


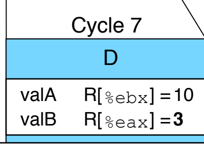


Data Hazard: Insert 3 NOP

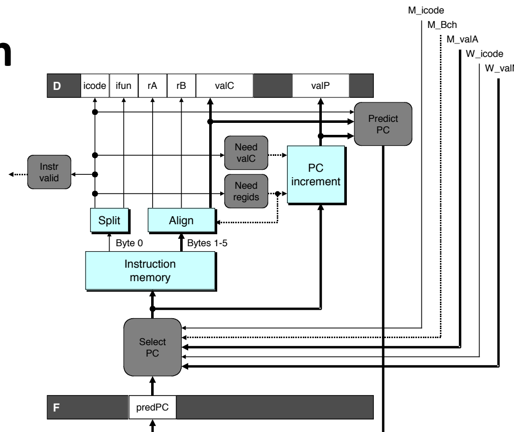
```
# demo-h3.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: nop
0x00f: addl  %edx,%eax
0x011: ...
```







PC Prediction



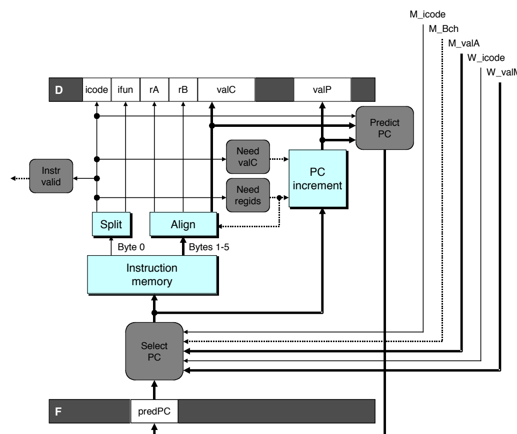
- We start fetch of new instruction after current one has completed fetch stage, so there's not enough time to determine next instruction
- Trick: try to *predict* which instruction will be executed next, and recover if prediction incorrect

Y86 Prediction Strategy

- **Instructions that don't transfer control**
  - Predict next PC to be valP
  - Always reliable
- **Call and Unconditional Jumps**
  - Predict next PC to be valC (destination)
  - Always reliable
- **Conditional Jumps: *Control Hazard***
  - Predict next PC to be valC (destination)
  - Only correct if branch is taken (**typically correct 60% of the time**)
- **Return instruction: *Control Hazard***
  - Don't try to predict!

Sewanee: The University of the South

Misprediction Recovery



Mispredicted Jump
- Sees branch flag once instruction reaches memory stage
- Correct "fall-through" PC is in valP

Return Instruction
- Gets return PC when `ret` reaches write-back stage

Branch Misprediction Example

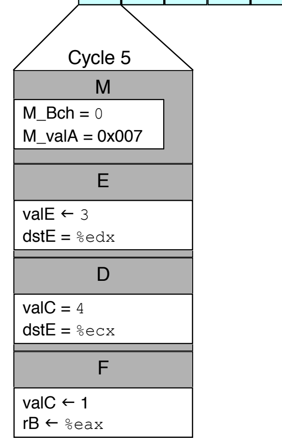
demo-j.ys

```
0x000:      xorl %eax,%eax
0x002:      jne  t                # Not taken
0x007:      irmovl $1, %eax       # Fall through
0x00d:      nop
0x00e:      nop
0x00f:      nop
0x010:      halt
0x011:  t:  irmovl $3, %edx       # Target (Should not execute)
0x017:      irmovl $4, %ecx       # Should not execute
0x01d:      irmovl $5, %edx       # Should not execute
```

Sewanee: The University of the South

Branch Misprediction Trace

# demo-j	1	2	3	4	5	6	7	8	9
0x000: xorl %eax,%eax	F	D	E	M	W				
0x002: jne t # Not taken		F	D	E	M	W			
0x011: t: irmovl $3, %edx # Target			F	D	E	M	W		
0x017: irmovl $4, %ecx # Target+1				F	D	E	M	W	
0x007: irmovl $1, %eax # Fall Through					F	D	E	M	W



Incorrectly execute two instructions at branch target!

Return Example

demo-ret.ys

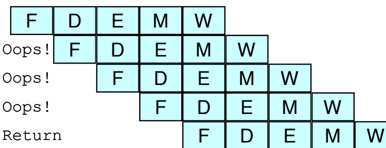
```
0x000:      irmovl Stack,%esp    # Initialize stack pointer
0x006:      nop                  # Avoid hazard on %esp
0x007:      nop
0x008:      nop
0x009:      call p               # Procedure call
0x00e:      irmovl $5,%esi       # Return point
0x014:      halt
…
0x020: p:   nop                  # procedure
0x021:      nop
0x022:      nop
0x023:      ret
0x024:      irmovl $1,%eax       # Should not be executed
0x02a:      irmovl $2,%ecx       # Should not be executed
0x030:      irmovl $3,%edx       # Should not be executed
0x036:      irmovl $4,%ebx       # Should not be executed
```

Requires many NOP to avoid data hazards

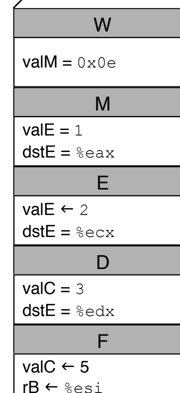
Incorrect Return Example

```
# demo-ret

0x023:    ret
0x024:    irmovl $1,%eax # Oops!
0x02a:    irmovl $2,%ecx # Oops!
0x030:    irmovl $3,%edx # Oops!
0x00e:    irmovl $5,%esi # Return
```



Incorrectly executes 3 instructions following `ret`



Pipeline Summary

- **Concept**
  - Break instruction execution into 5 stages
  - Run instructions through in pipelined mode
- **Limitations**
  - Can't handle dependencies between instructions when instructions follow too closely
  - Data hazards
    - One instruction writes register, later one reads it
  - Control hazards
    - Instruction sets PC in way that pipeline did not predict correctly
    - Mispredicted branch and return
- **Fixing the Pipeline**

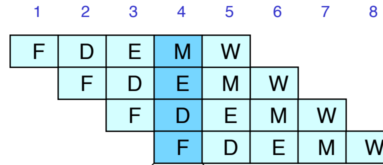
  We'll do that next!

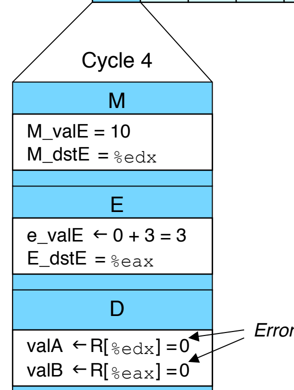
Pipelining Hazards

- **Data Hazards**
  - Instruction having register R as *source* follows soon after instruction having register R as *destination*
  - Very common condition; how to keep from slowing down pipeline
- **Control Hazards**
  - Mispredicted conditional branch
    - Our policy predicts all branches as being taken
    - Naïve pipeline executes two extraneous instructions
  - Getting return address for `ret` instruction
    - Naïve pipeline executes three extraneous instructions
- **Structural Hazards**
  - Contention for bus by Fetch and Memory units
  - Modern processor/cache design alleviates these hazards
- **Making Sure It Really Works**
  - What if multiple special cases happen simultaneously?

Data Hazard

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```



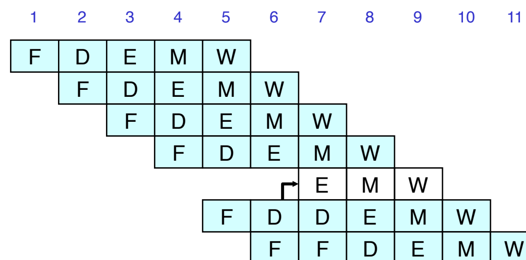


Naïve fix: insert three NOP instructions before addl instruction.

Sewanee: The University of the South

Stalling for Data Hazard

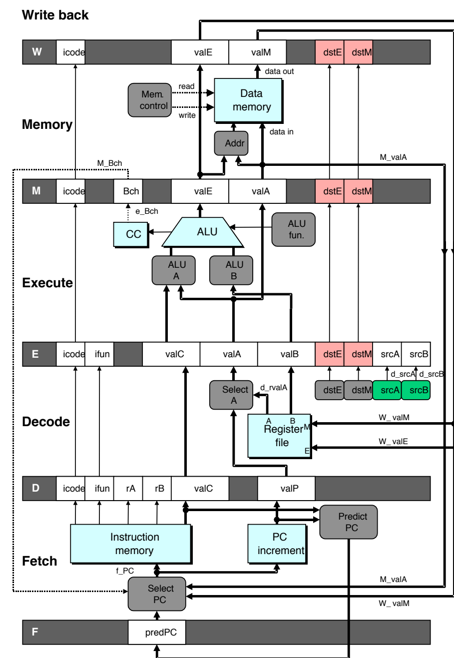
```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
        bubble
0x00e: addl %edx,%eax
0x010: halt
```



- If instruction that reads register follows too closely after one that writes register, slow pipeline down
- Hold instruction in decode
- Dynamically inject nop into execute stage

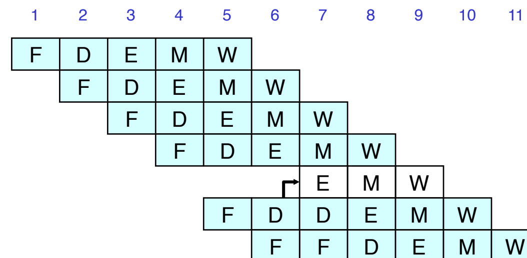
Stall Condition

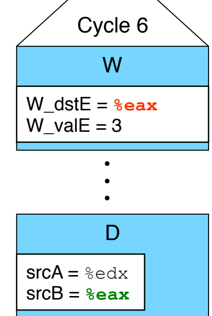
- **Source Registers**
  - **srcA** and **srcB** of current instruction in decode stage
- **Destination Registers**
  - **dstE** and **dstM** fields
  - Instructions in execute, memory, and write-back stages
- **Special Case**
  - Don't stall for register ID 8
    - Indicates absence of register operand



Detecting Stall Condition

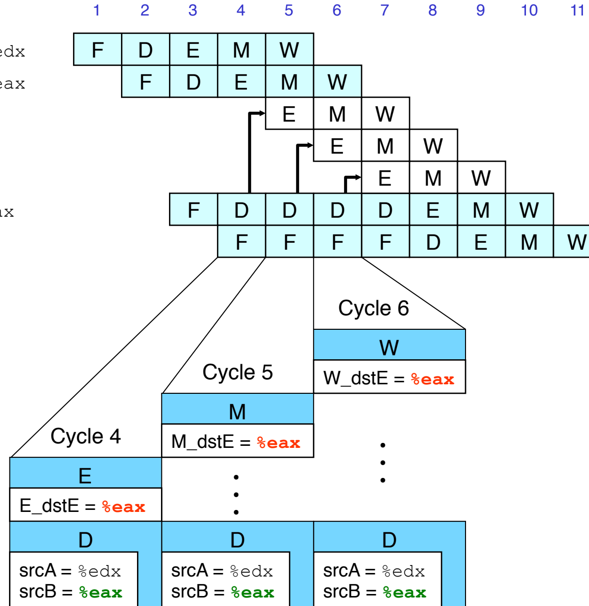
```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
       bubble
0x00e: addl %edx,%eax
0x010: halt
```





Stalling X3

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
       bubble
       bubble
       bubble
0x00c: addl %edx,%eax
0x00e: halt
```



What Happens When Stalling?

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```

Cycle 8	
Write Back	***bubble***
Memory	***bubble***
Execute	0x00c: addl %edx,%eax
Decode	0x00e: halt
Fetch	

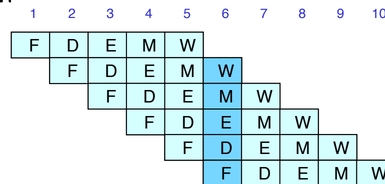
- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- *Bubbles* injected into execute stage
  - Like dynamically generated nop's
  - Move through later stages

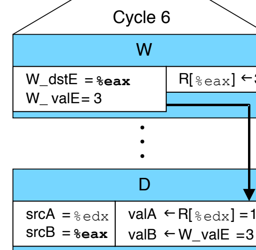
Data Forwarding

- **In Naïve Pipeline design**
  - Register isn't written until completion of write-back stage
  - Source operands read from register file in decode stage
    - Needs to be in register file at start of stage
- **Observation**
  - Value to be written back is generated in execute or memory stage
- **Trick**
  - Pass value directly from generating instruction to decode stage
  - Needs to be available at *end* of decode stage

Data Forwarding Example

```
# demo-h2.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```

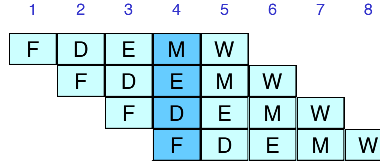


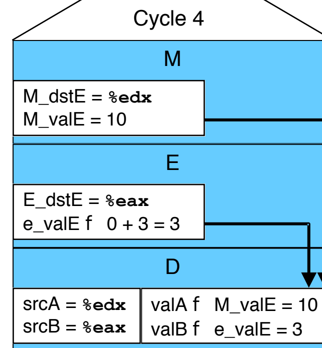


- `irmovl` in write-back stage in cycle 6
- Destination value in W pipeline register
- Forward to decode stage as **valB**

Data Forwarding Example #2

```
# demo-h0.ys
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: addl %edx,%eax
0x00e: halt
```





Register %edx

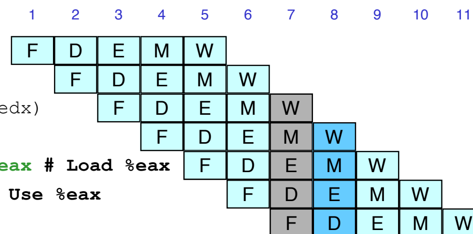
- Generated by ALU during previous cycle
- Forward from memory as valA

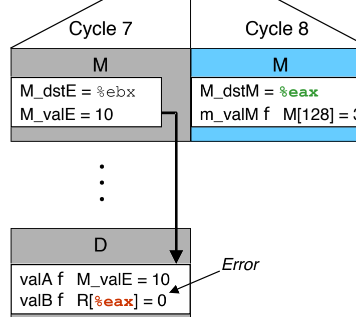
Register %eax

- Value just generated by ALU
- Forward from execute as valB

Limitation of Forwarding

```
# demo-luh.ys
0x000: irmovl $128,%edx
0x006: irmovl  $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```





Load-use dependency

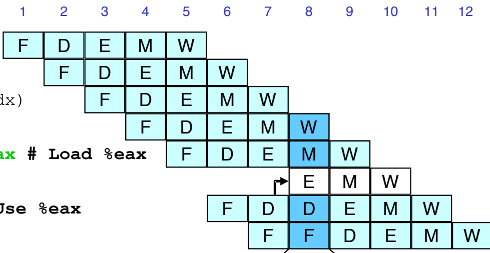
- Value needed by end of decode stage in cycle 7
- Value read from memory in memory stage of cycle 8

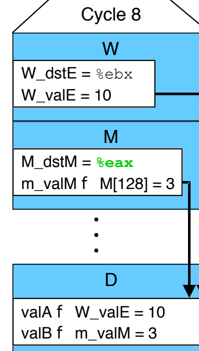
Sewanee: The University of the South

Avoiding Load/Use Hazard

```
# demo-luh.ys

0x000: irmovl $128,%edx
0x006: irmovl  $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
       bubble
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```





- **Stall** using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

Sewanee: The University of the South

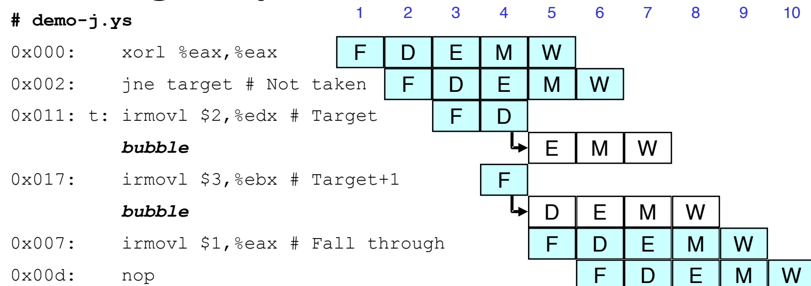
Branch Misprediction Example

```
demo-j.ys

0x000:     xorl %eax,%eax
0x002:     jne  t              # Not taken
0x007:     irmovl $1, %eax     # Fall through
0x00d:     nop
0x00e:     nop
0x00f:     nop
0x010:     halt
0x011: t:  irmovl $3, %edx     # Target (Should not execute)
0x017:     irmovl $4, %ecx     # Should not execute
0x01d:     irmovl $5, %edx     # Should not execute
```

Should only execute first 87 instructions

Handling Misprediction



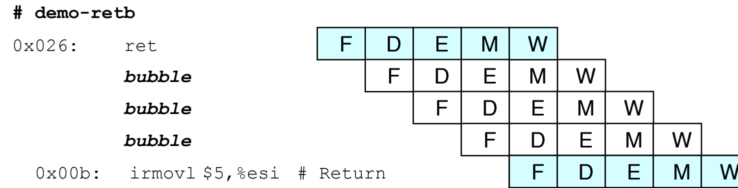
- Predict branch is taken
  - Fetch 2 instructions at target
- *Cancel* when mispredicted
  - Detect branch not-taken in execute stage
  - On following cycle, replace instructions in execute and decode by bubbles
  - No side effects have occurred yet

Return Example

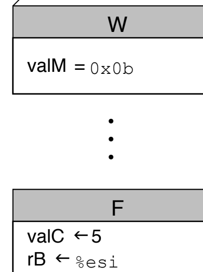
```
0x000:    irmovl Stack,%esp  # Initialize stack pointer
0x006:    call p             # Procedure call
0x00b:    irmovl $5,%esi     # Return point
0x011:    halt
0x020: .pos 0x20
0x020: p: irmovl $-1,%edi    # procedure
0x026:    ret
0x027: b: irmovl $1,%eax     # Should not be executed
0x02d:    irmovl $2,%ecx     # Should not be executed
0x033:    irmovl $3,%edx     # Should not be executed
0x039:    irmovl $4,%ebx     # Should not be executed
0x100: .pos 0x100
0x100: Stack:                # Stack: Stack pointer
```

Previously, pipeline incorrectly executed three additional instructions

Corrected Return Example



1. As `ret` passes through pipeline, stall at the fetch stage
2. Inject bubble into decode stage
3. Release stall when write-back stage is reached



Sewanee: The University of the South

Pipeline Summary

- **Data Hazards**
  - Most can be handled by forwarding - **no performance penalty**!
  - Load/use hazard requires one cycle stall
- **Control Hazards**
  - When mispredicted branch detected, cancel instructions
    - Two clock cycles wasted
  - Stall fetch stage when `ret` passes through pipeline
    - Three clock cycles wasted
- **Structural Hazards**
  - Fetching instruction collides with reading/writing data to memory
  - Not a problem on most modern processors: on-chip cache accesses data/instructions separately