

CSCE 230J
Computer Organization

Processor Architecture V: Making the Pipelined Implementation Work

Dr. Steve Goddard
goddard@cse.unl.edu

<http://cse.unl.edu/~goddard/Courses/CSCE230J>

Giving credit where credit is due

- **Most of slides for this lecture are based on slides created by Dr. Bryant, Carnegie Mellon University.**
- **I have modified them and added new slides.**

Overview

Make the pipelined processor work!

Data Hazards

- Instruction having register R as source follows shortly after instruction having register R as destination
- Common condition, don't want to slow down pipeline

Control Hazards

- Mispredict conditional branch
 - Our design predicts all branches as being taken
 - Naïve pipeline executes two extra instructions
- Getting return address for `ret` instruction
 - Naïve pipeline executes three extra instructions

Making Sure It Really Works

- What if multiple special cases happen simultaneously?

3

Pipeline Stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

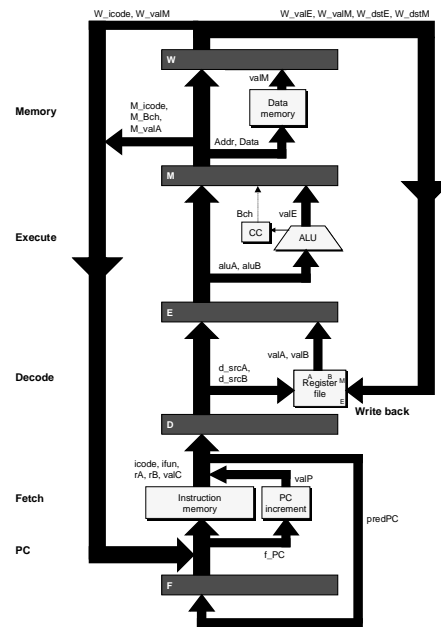
- Operate ALU

Memory

- Read or write data memory

Write Back

- Update register file



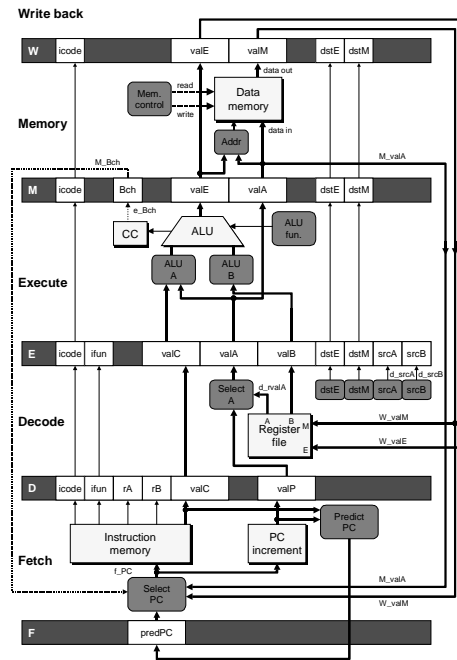
4

PIPE- Hardware

- Pipeline registers hold intermediate values from instruction execution

Forward (Upward) Paths

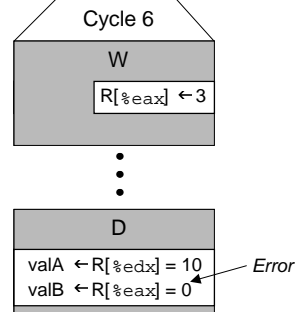
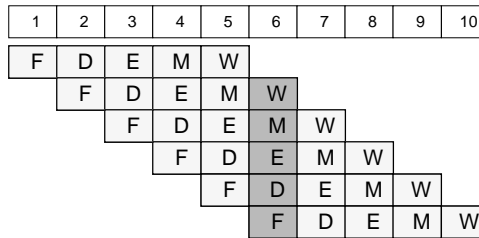
- Values passed from one stage to next
- Cannot jump past stages
 - e.g., valC passes through decode



5

Data Dependencies: 2 Nop's

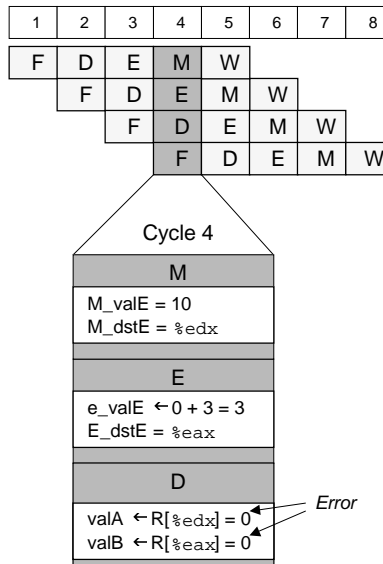
```
# demo-h2.y
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: nop
0x00e: addl %edx,%eax
0x010: halt
```



6

Data Dependencies: No Nop

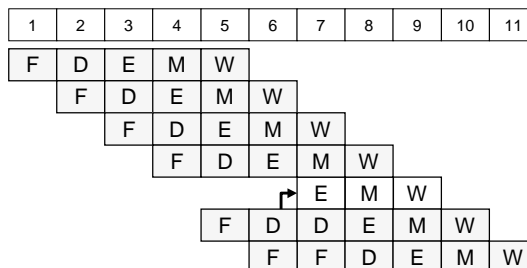
demo-h0.y _s
0x000: irmovl \$10,%edx
0x006: irmovl \$3,%eax
0x00c: addl %edx,%eax
0x00e: halt



7

Stalling for Data Dependencies

demo-h2.y _s
0x000: irmovl \$10,%edx
0x006: irmovl \$3,%eax
0x00c: nop
0x00d: nop
<i>bubble</i>
0x00e: addl %edx,%eax
0x010: halt



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

8

Stall Condition

Source Registers

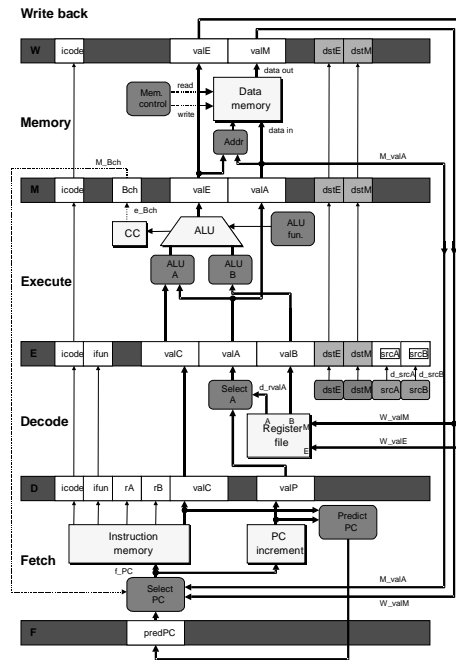
- srcA and srcB of current instruction in decode stage

Destination Registers

- dstE and dstM fields
- Instructions in execute, memory, and write-back stages

Special Case

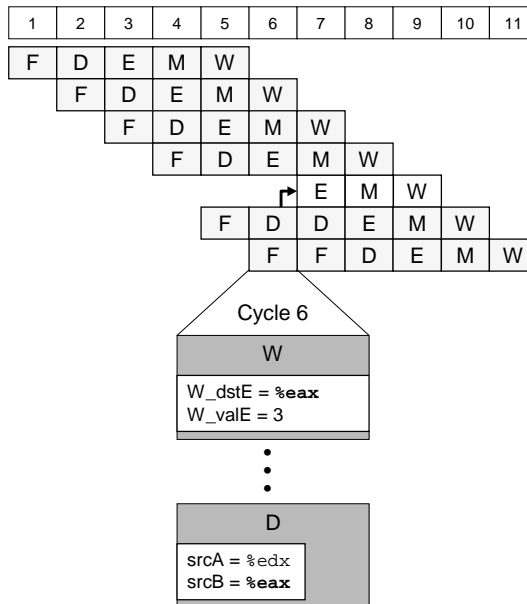
- Don't stall for register ID 8
 - Indicates absence of register operand



9

Detecting Stall Condition

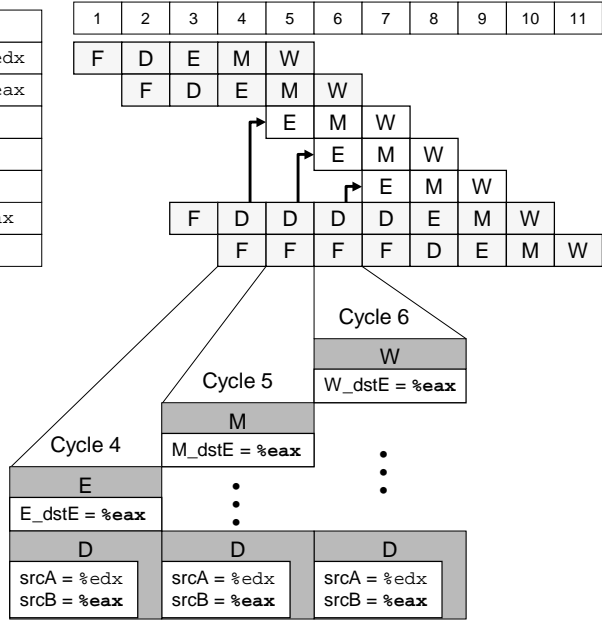
```
# demo-h2.y
0x000: irmovl $10,%edx
0x006: irmovl $3,%eax
0x00c: nop
0x00d: nop
bubble
0x00e: addl %edx,%eax
0x010: halt
```



10

Stalling X3

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



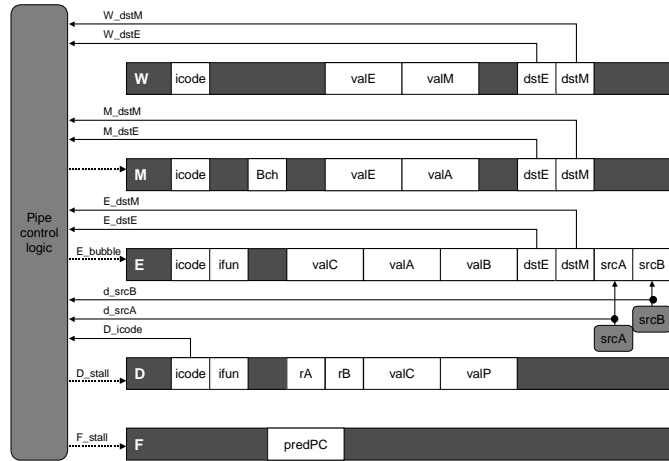
What Happens When Stalling?

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

	Cycle 8
Write Back	<i>bubble</i>
Memory	<i>bubble</i>
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

Implementing Stalling

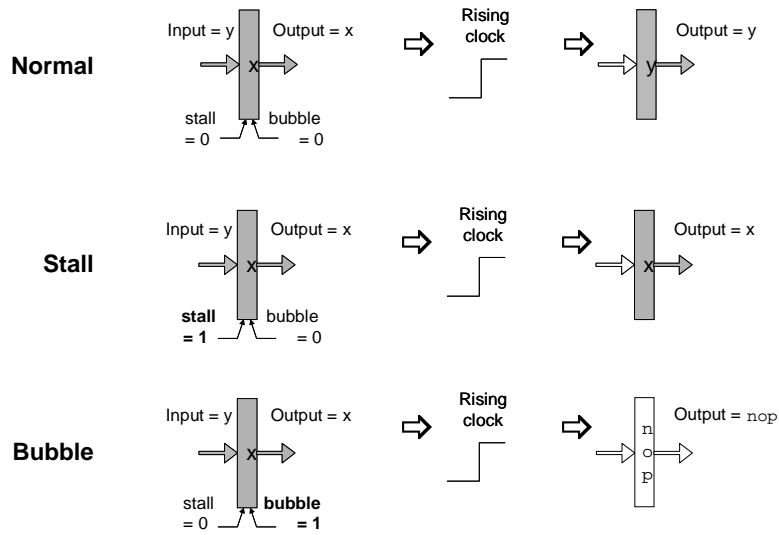


Pipeline Control

- Combinational logic detects stall condition
- Sets mode signals for how pipeline registers should update

13

Pipeline Register Modes



14

Data Forwarding

Naïve Pipeline

- 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 generated in execute or memory stage

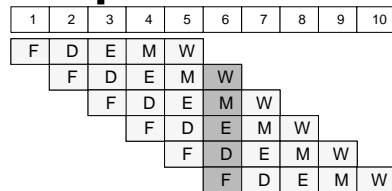
Trick

- Pass value directly from generating instruction to decode stage
- Needs to be available at end of decode stage

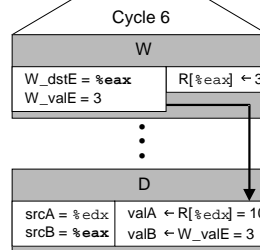
15

Data Forwarding Example

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



- `irmovl` in write-back stage
- Destination value in W pipeline register
- Forward as `valB` for decode stage



16

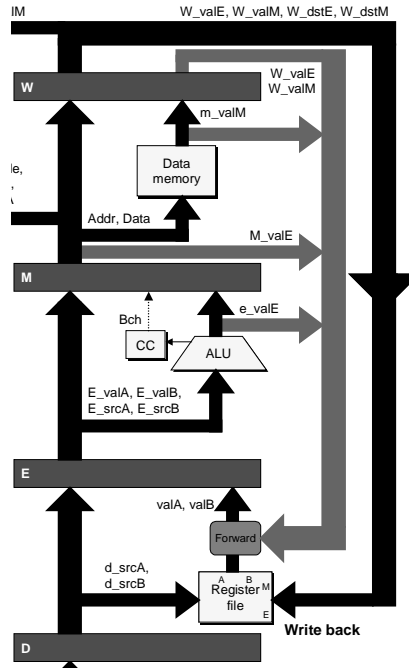
Bypass Paths

Decode Stage

- Forwarding logic selects valA and valB
- Normally from register file
- Forwarding: get valA or valB from later pipeline stage

Forwarding Sources

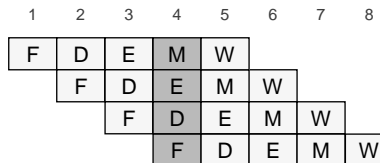
- Execute: valE
- Memory: valE, valM
- Write back: valE, valM



17

Data Forwarding Example #2

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

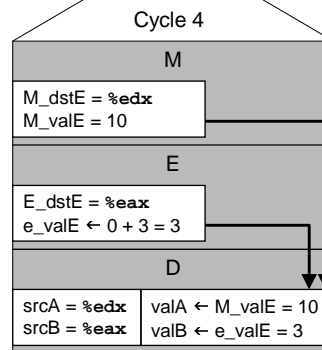


Register %edx

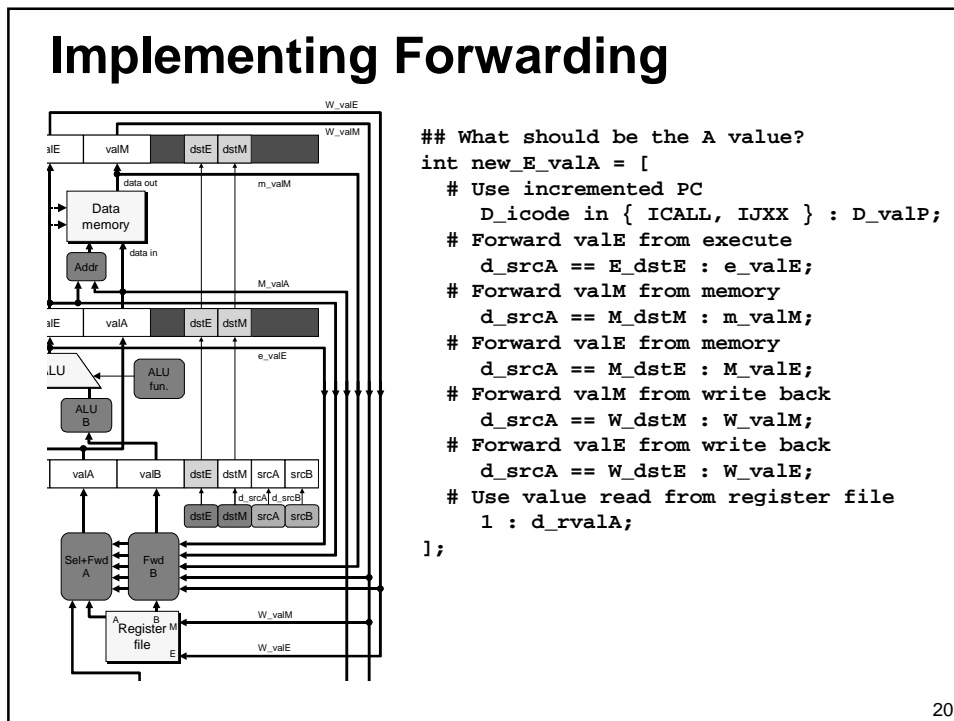
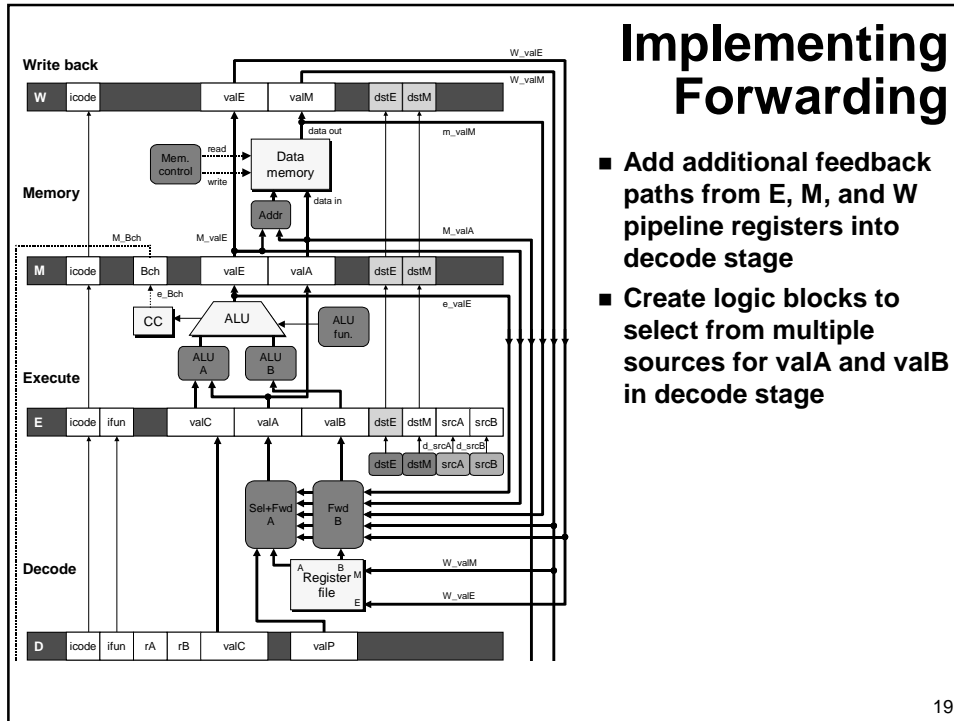
- Generated by ALU during previous cycle
- Forward from memory as valA

Register %eax

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



18



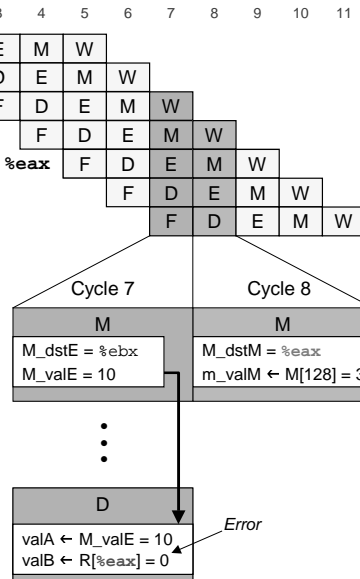
Limitation of Forwarding

```

# demo-luh.js
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



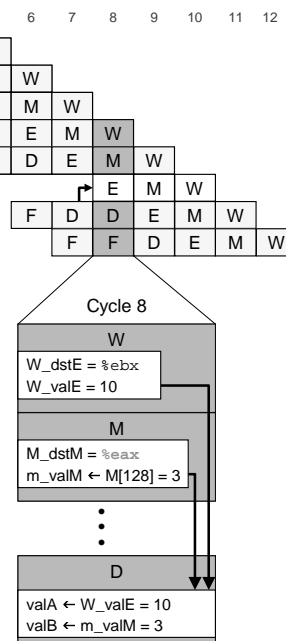
21

Avoiding Load/Use Hazard

```

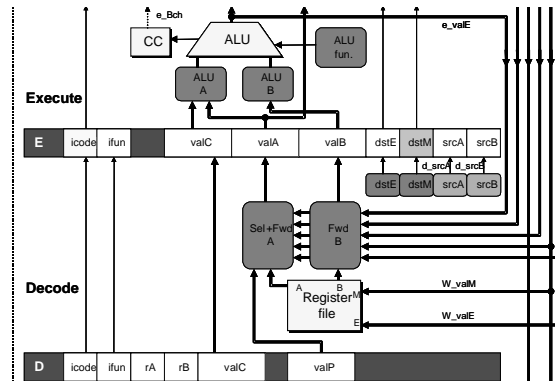
# demo-luh.js
0x000: irmovl $128,%edx
0x006: irmovl $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
      bubble
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
    
```

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



22

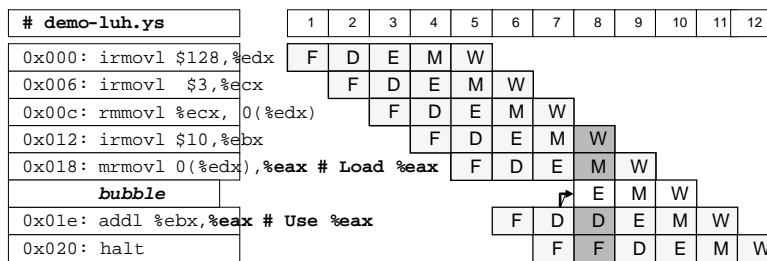
Detecting Load/Use Hazard



Condition	Trigger
Load/Use Hazard	E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }

23

Control for Load/Use Hazard



- Stall instructions in fetch and decode stages
- Inject bubble into execute stage

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal

24

Branch Misprediction Example

demo-j.js

```

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 8 instructions

25

Handling Misprediction

# demo-j.js	1	2	3	4	5	6	7	8	9	10
0x000: xorl %eax,%eax	F	D	E	M	W					
0x002: jne target # Not taken	F	D	E	M	W					
0x011: t: irmovl \$2,%edx # Target			F	D						
<i>bubble</i>					E	M	W			
0x017: irmovl \$3,%ebx # Target+1				F						
<i>bubble</i>					D	E	M	W		
0x007: irmovl \$1,%eax # Fall through					F	D	E	M	W	
0x00d: nop						F	D	E	M	W

Predict branch as taken

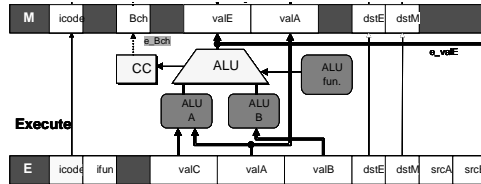
- Fetch 2 instructions at target

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

26

Detecting Mispredicted Branch



Condition	Trigger
Mispredicted Branch	E_icode = IJXX & !e_Bch

27

Control for Misprediction

```

# demo-j.js
1 2 3 4 5 6 7 8 9 10
0x000: xorl %eax,%eax      F D E M W
0x002: jne target # Not taken F D E M W
0x011: t: irmovl $2,%edx # Target
      bubble
0x017: irmovl $3,%ebx # Target+1
      bubble
0x007: irmovl $1,%eax # Fall through
0x00d: nop
  
```

Condition	F	D	E	M	W
Mispredicted Branch	normal	bubble	bubble	normal	normal

28

Return Example

demo-retb.js

```

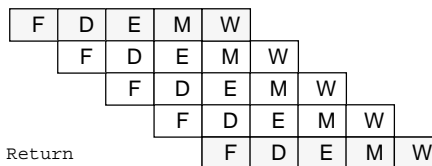
0x000:   irmovl Stack,%esp # Initialize stack pointer
0x006:   call p             # Procedure call
0x00b:   irmovl $5,%esi   # Return point
0x011:   halt
0x020:   .pos 0x20
0x020: p: irmovl $-1,%edi # procedure
0x026:   ret
0x027:   irmovl $1,%eax   # Should not be executed
0x02d:   irmovl $2,%ecx   # 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 executed three additional instructions

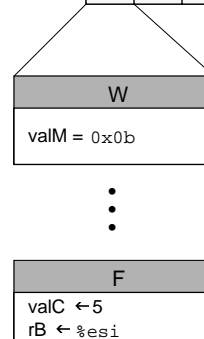
29

Correct Return Example

demo-retb
0x026: ret
bubble
bubble
bubble
0x00b: irmovl \$5,%esi # Return

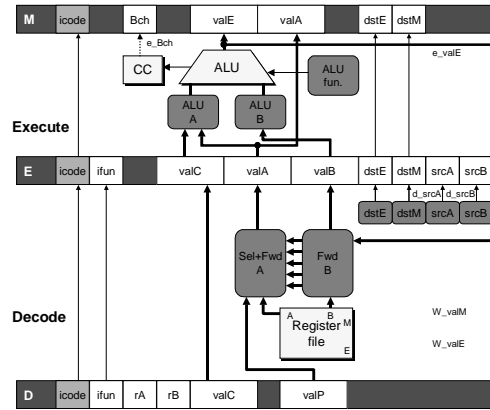


- As `ret` passes through pipeline, stall at fetch stage
 - While in decode, execute, and memory stage
- Inject bubble into decode stage
- Release stall when reach write-back stage



30

Detecting Return

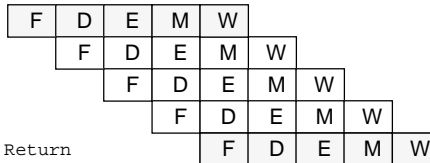


Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }

31

Control for Return

```
# demo-retb
0x026:  ret
      bubble
      bubble
      bubble
0x00b:  irmovl $5,%esi # Return
```



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal

32

Special Control Cases

Detection

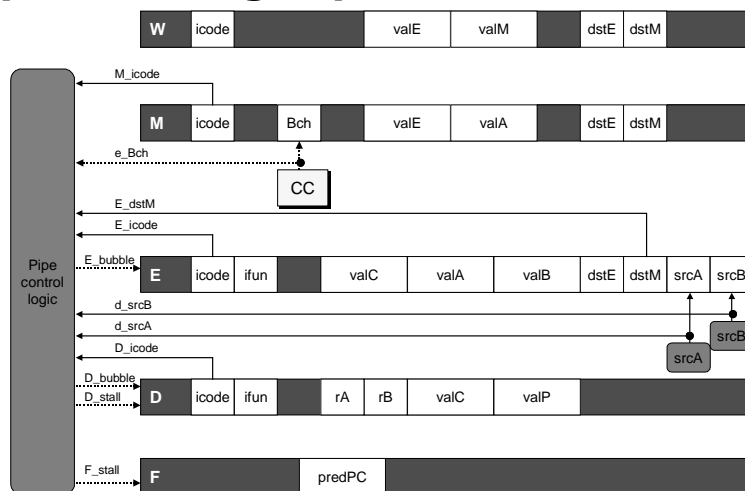
Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode = IJXX & !e_Bch

Action (on next cycle)

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal

33

Implementing Pipeline Control



- Combinational logic generates pipeline control signals
- Action occurs at start of following cycle

34

Initial Version of Pipeline Control

```

bool F_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB } ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

bool D_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB };

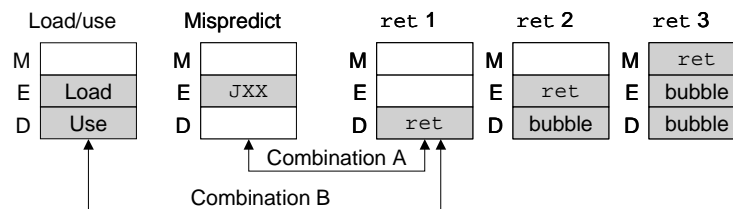
bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Bch) ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode };

bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Bch) ||
    # Load/use hazard
    E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB};

```

35

Control Combinations



- Special cases that can arise on same clock cycle

Combination A

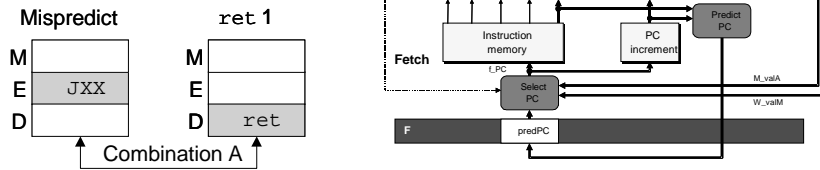
- Not-taken branch
- `ret` instruction at branch target

Combination B

- Instruction that reads from memory to `%esp`
- Followed by `ret` instruction

36

Control Combination A



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

- Should handle as mispredicted branch
- Stalls F pipeline register
- But PC selection logic will be using M_valM anyhow

37

Control Combination B



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	stall	bubble + stall	bubble	normal	normal

- Would attempt to bubble *and* stall pipeline register D
- Signaled by processor as pipeline error

38

Handling Control Combination B



Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
<i>Combination</i>	<i>stall</i>	<i>stall</i>	<i>bubble</i>	<i>normal</i>	<i>normal</i>

- Load/use hazard should get priority
- ret instruction should be held in decode stage for additional cycle

39

Corrected Pipeline Control Logic

```

bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Bch) ||
    # Stalling at fetch while ret passes through pipeline
    IRET in { D_icode, E_icode, M_icode }
    # but not condition for a load/use hazard
    && !(E_icode in { IMRMOVL, IPOPL }
        && E_dstM in { d_srcA, d_srcB });
    
```

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
<i>Combination</i>	<i>stall</i>	<i>stall</i>	<i>bubble</i>	<i>normal</i>	<i>normal</i>

- Load/use hazard should get priority
- ret instruction should be held in decode stage for additional cycle

40

Pipeline Summary

Data Hazards

- Most handled by forwarding
 - No performance penalty
- Load/use hazard requires one cycle stall

Control Hazards

- Cancel instructions when detect mispredicted branch
 - Two clock cycles wasted
- Stall fetch stage while `ret` passes through pipeline
 - Three clock cycles wasted

Control Combinations

- Must analyze carefully
- First version had subtle bug
 - Only arises with unusual instruction combination