Chapter 4: Pipelined Processors

Topics

- Stalling
- Branch Misprediction Example

Announcements

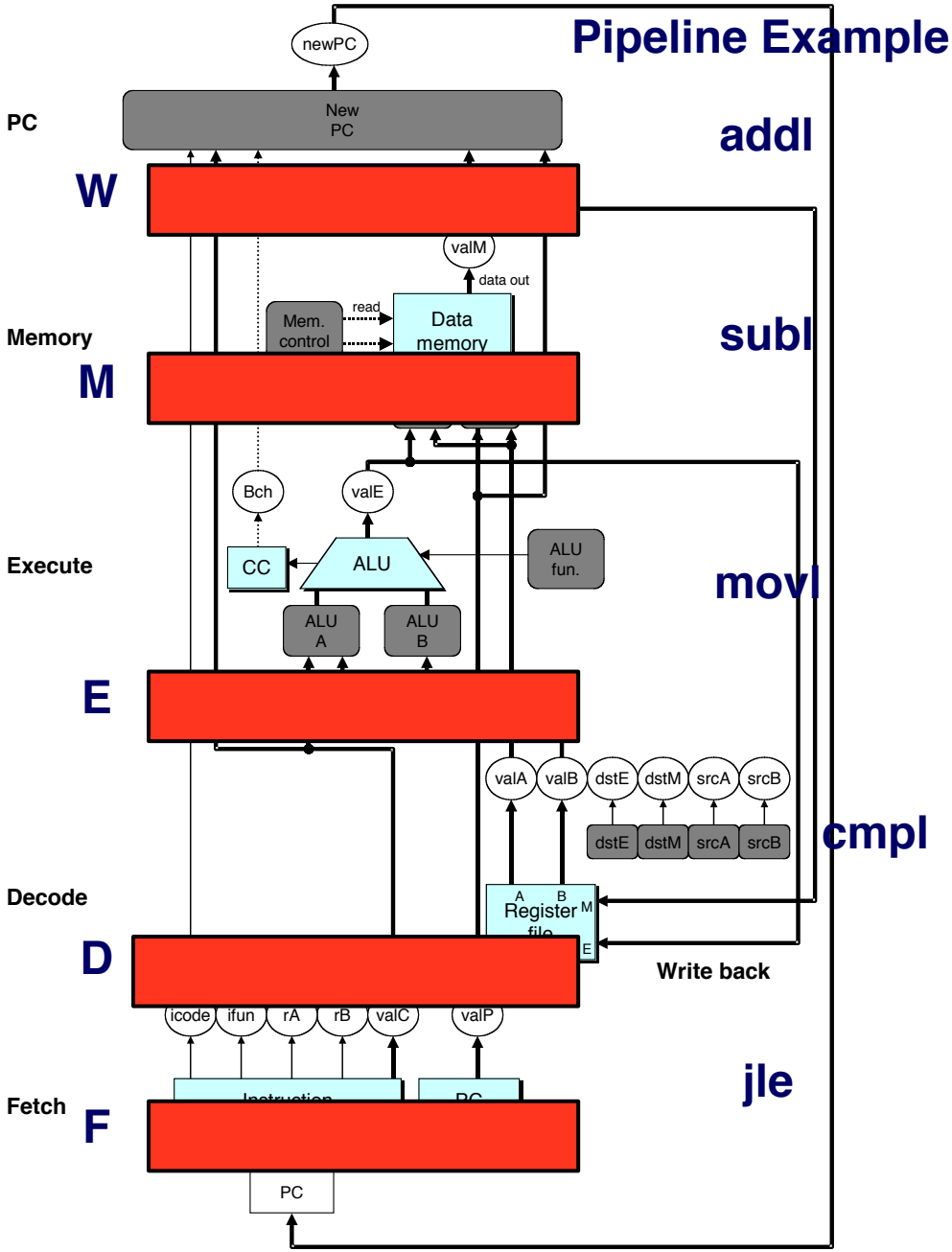
- Buffer Lab is due next Monday Oct 27 by 8 am
  - Sign up for time slots later this week
- Recitation Exercises #4 released next Monday, due in a week
- Midterm #2 approximately the week of Nov 10
- Essential that you read the textbook in detail & do the practice problems
  - Read Chapter 4, but Skip 4.2, 4.3.4, 4.5.9-4.5.11 (skip the PIPE implementation), 4.5.13. Overall, skipping these sections will save you about 50 pages of reading

Recap

Traced execution of a single instruction (`addl`, `rmmovl`) through 6 stages of CPU

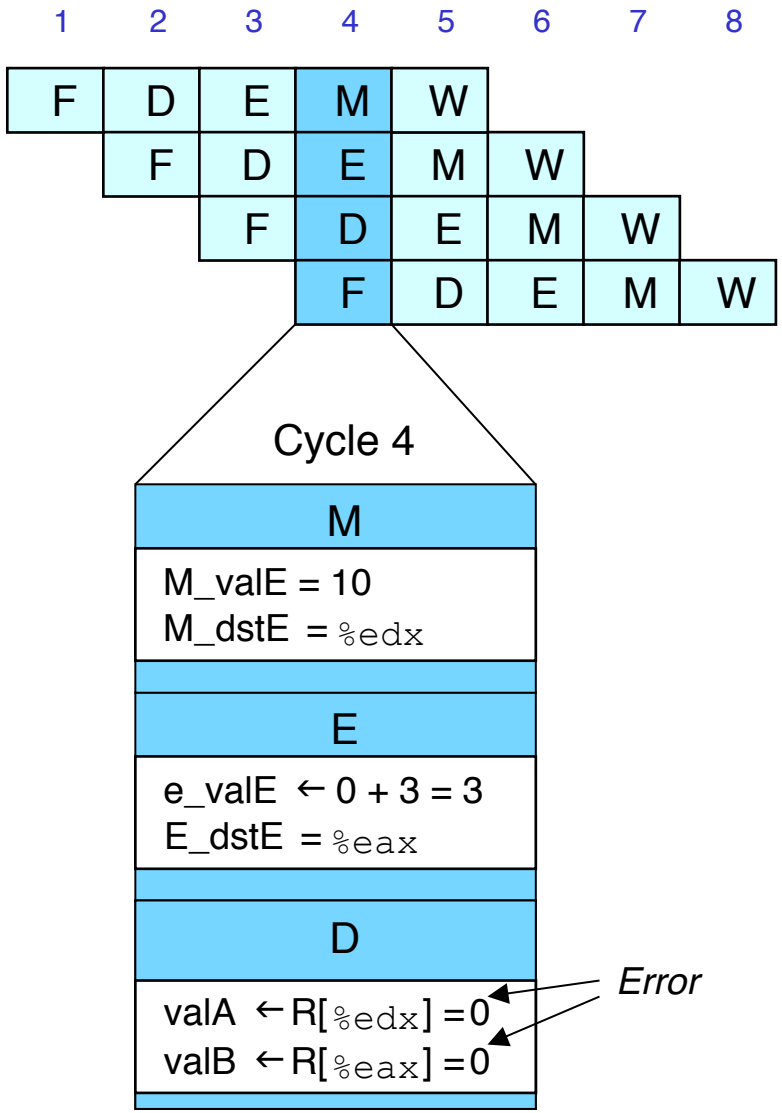
Insert registers between each stage to create *pipeline*

- **Increases instruction processing rate of CPU**
- **Limitations: nonuniform delays, register delays**
- **Issue: Data dependency/ hazard**
- **Issue: Branch misprediction**



Data Dependencies: No Nop

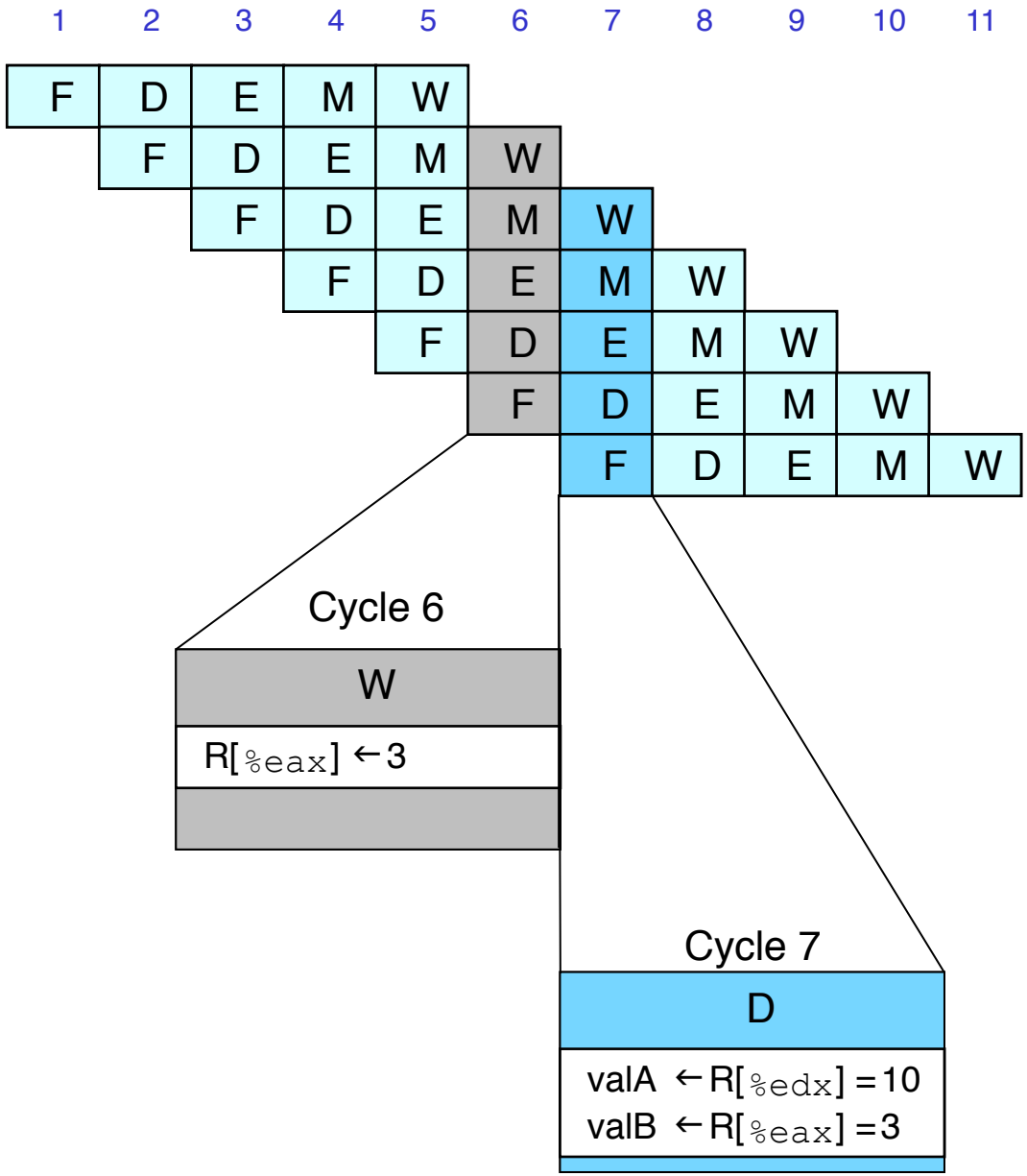
```
# demo-h0.ys
 0x000: irmovl $10,%edx
 0x006: irmovl  $3,%eax
 0x00c: addl %edx,%eax
0x00e: halt
```



Assume all registers initialized to zero

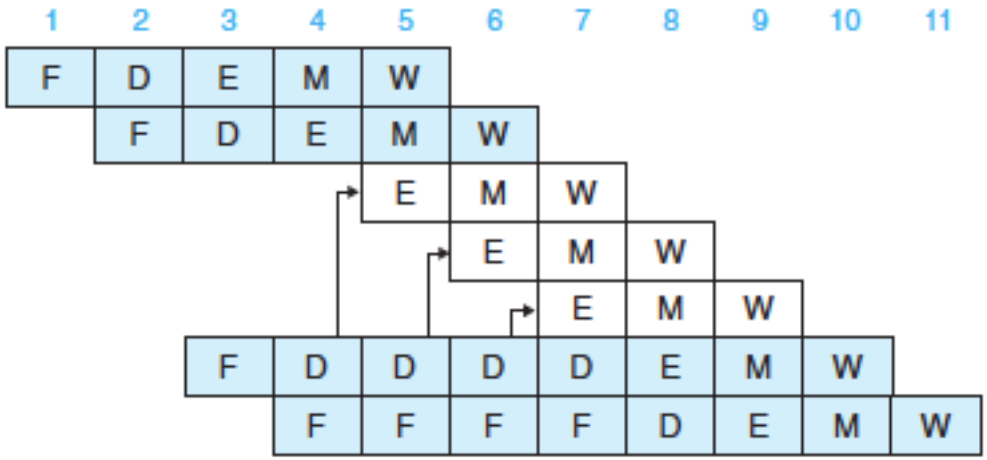
Data Dependencies: 3 Nop's

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



Stalling an Instruction

```
# prog4
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
       bubble
       bubble
       bubble
0x00c: addl %edx,%eax
0x00e: halt
```



CPU hardware recognizes a data dependency/hazard and stalls an instruction until results are ready

- **In our example, `addl` instruction depends on %edx and %eax**
- **These are not set until the Writeback stages of each irmovl are done**
- **So stall `addl` instruction at the Decode stage, introducing bubbles or effectively nops**

Branch Misprediction Example

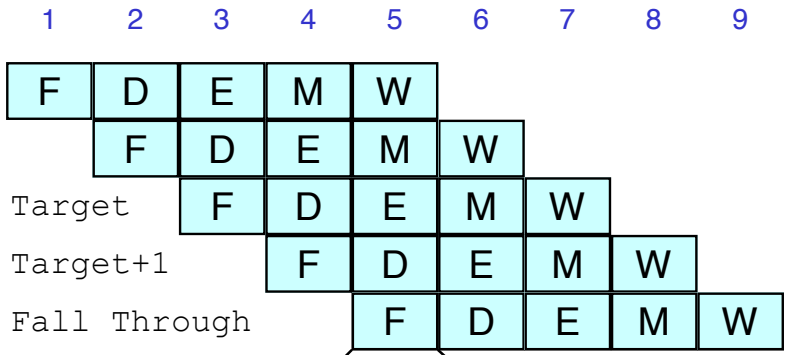
demo-j.ys

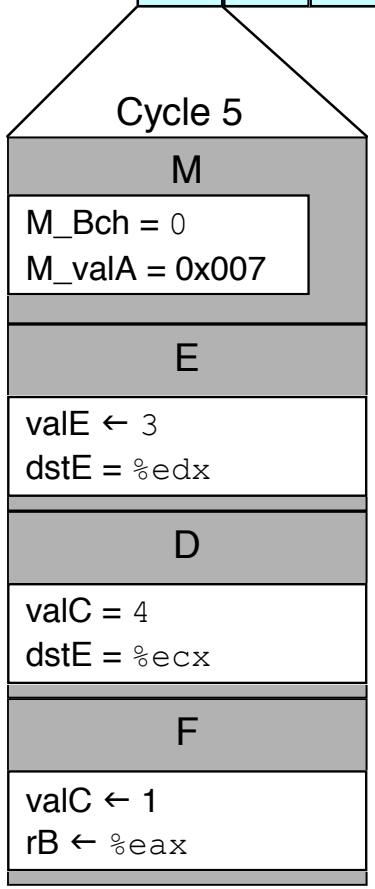
```
0x000:     xorl %eax,%eax      # %eax always zero
0x002:     jne  t              # Not taken
0x007:     irmovl $1, %eax     # Fall through
0x00d:     nop
0x00e:     nop
0x00f:     nop
0x010:     halt
0x011: t:  irmovl $3, %edx     # Target (Should not execute)
0x017:     irmovl $4, %ecx     # Should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

- Should only execute first 7 instructions

Branch Misprediction Trace

```
# demo-j
0x000:     xorl %eax,%eax
0x002:     jne t # Not taken
0x011: t:  irmovl $3, %edx # Target
0x017:     irmovl $4, %ecx # Target+1
0x007:     irmovl $1, %eax # Fall Through
```





- **Incorrectly execute multiple instructions at branch target**
- **CPU hardware realizes a mistake has occurred in branch prediction, and ignores the results of speculative execution, makes the correct jump and starts refilling the pipeline**

Return Example

demo-ret.ys

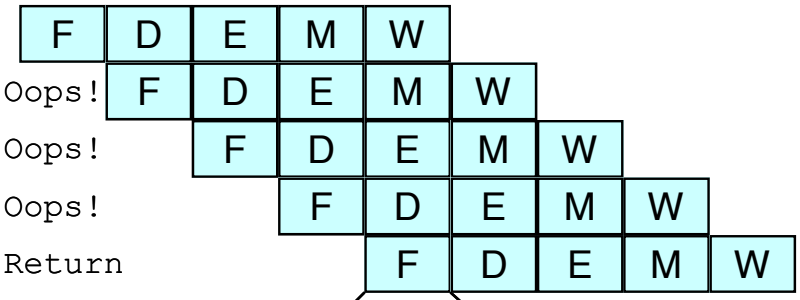
```
0x000:    irmovl Stack,%esp  # Initialize stack pointer
0x006:    nop                # Avoid hazard on %esp
0x007:    nop
0x008:    nop
0x009:    call p             # Procedure call
0x00e:    irmovl $5,%esi     # Return point
0x014:    halt
0x020: .pos 0x20
0x020: p: nop                # procedure
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
0x100: .pos 0x100
0x100: Stack:                # Stack: Stack pointer
```

- Require lots of nops to avoid data hazards

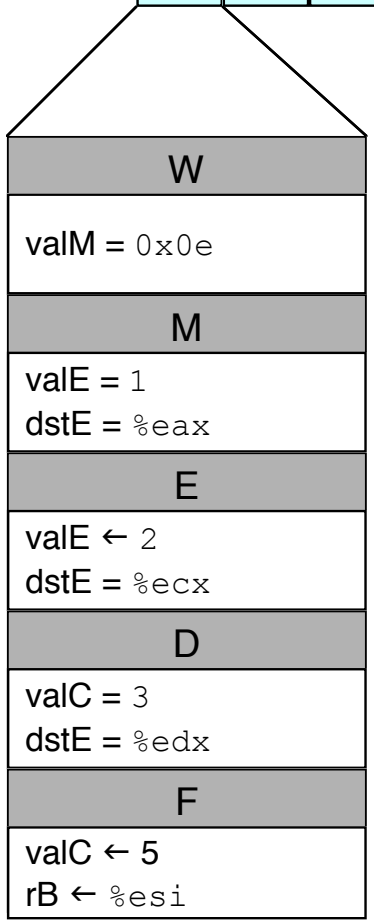
Incorrect Return Example

```
# 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 execute 3 instructions following `ret`**



Supplementary Slides

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 dependencies
  - One instruction writes register, later one reads it
- Control dependency
  - Instruction sets PC in way that pipeline did not predict correctly
  - Mispredicted branch and return

Fixing the Pipeline – see text

- Stalling – CPU sees dependencies & dynamically inserts nops
- Forwarding data directly to next stage(s) rather than wait for the clock

Pipeline Summary

More modern CPUs use out-of-order execution and parallel execution to achieve faster performance

- **Pipelining is still popular in embedded processors**