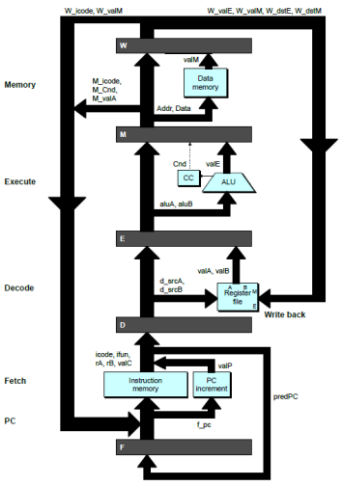
Pipeline Stages

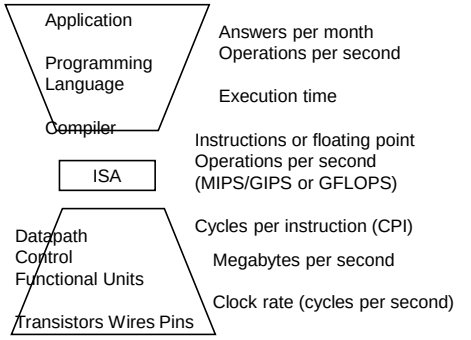
- Fetch
  - Select current PC
  - Read instruction
  - Compute incremented PC
- Decode
  - Read program registers
- Execute
  - Operate ALU
- Memory
  - Read or write data memory
- Write Back
  - Update register file



116

116

Metrics of Performance



117

Performance Metrics

- Throughput
- Execution time
- Instruction count
- Instruction rate (MIPS/GIPS)
- Cycles per instruction (CPI)
- Clock cycle time
- Clock rate – inverse of clock cycle time

118

Relating Performance Metrics

- CPU Time = Instructions X Average CPI X Cycle time

	Instr. Count	Avg. CPI	Clock rate
Program	x		
Compiler	x	X	
ISA	x	X	
Organization		X	X
Technology			X

119

Breakout

- Your program executes 100 million instructions. 50% of the instructions are ALU instructions that take 1 cycle each. 50% of the instructions load or store data from or to memory and take 3 cycles each. Your machine's clocks speed is 2 GHz. How long will your program take to execute?

Hazards

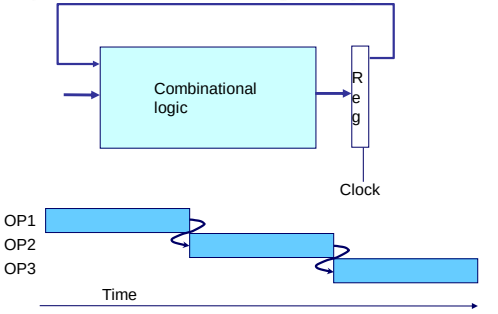
- Structural (e.g., instruction/data fetch)
- Data (e.g., add followed by sub reading dst register of add)
- Control (e.g., jeq)

Data Dependencies

```
1    irmovq $50,  %rax
2    addq   %rax, %rbx
3    mrmovq 100(%rbx),  %rdx
```

- Result from one instruction used as operand for another
  - **Read-after-write (RAW)** dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

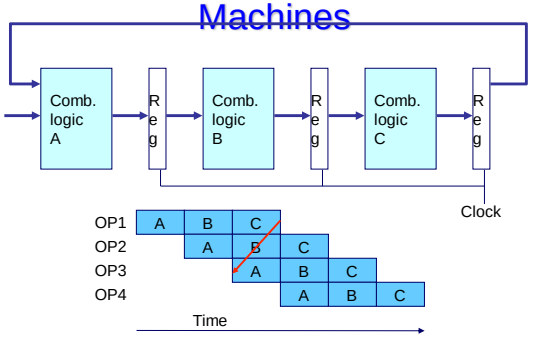
Data Dependencies in Single-Cycle Machines



In Single-Cycle Implementation:

- Each operation starts only after the previous operation finishes. Dependency always satisfied.

Data Dependencies in Pipeline Machines



Data Hazards happen when:

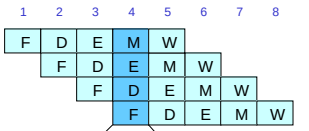
– Result does not feed back around in time for next operation

126

126

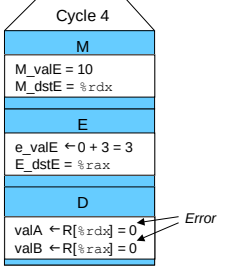
Data Dependencies: No Nop

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Remember registers get updated in the Write-back stage

addq reads wrong %rdx and %rax

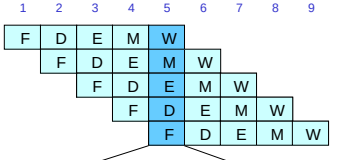


127

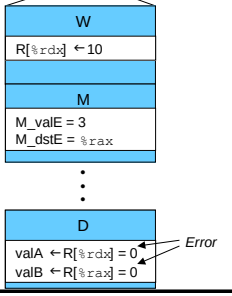
127

Data Dependencies: 1 Nop

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```



addq still reads wrong %rdx and %rax

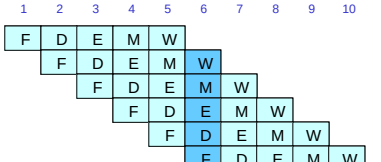


128

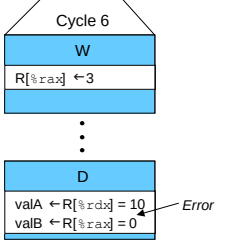
128

Data Dependencies: 2 Nop's

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



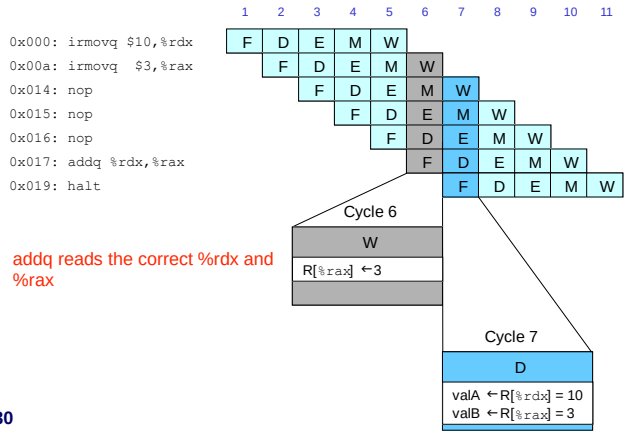
addq reads the correct %rdx, but %rax still wrong



129

129

Data Dependencies: 3 Nop's



addq reads the correct %rdx and %rax

130

130

Resolving Data Dependencies

- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a great idea
- Hardware mechanisms
  - Stalling
  - Forwarding
  - Out-of-order execution

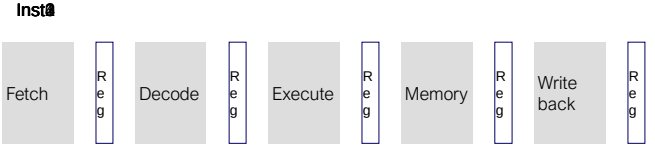
131

131

Hardware Generated Nops (Bubble and Stalling)

Can we have the hardware automatically generates a nop?

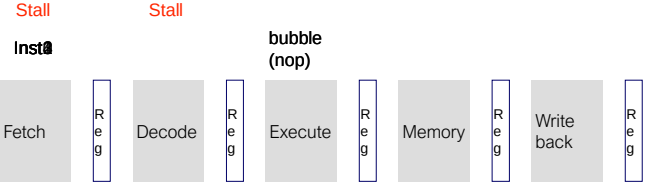
- Why is it good for the hardware to do so anyways?



132

132

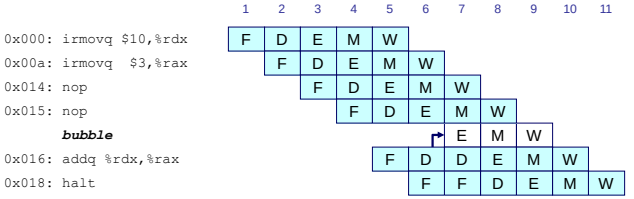
Stalling Illustration



133

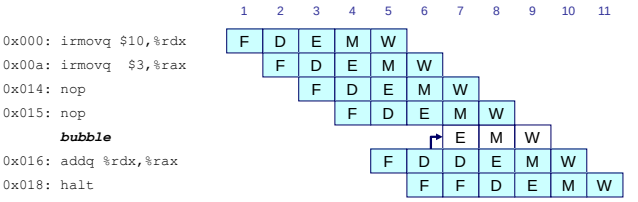
133

Stalling for Data Dependencies



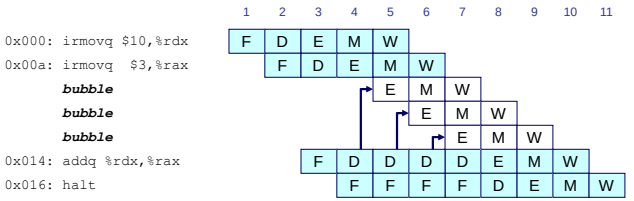
- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode

Detecting Stall Condition

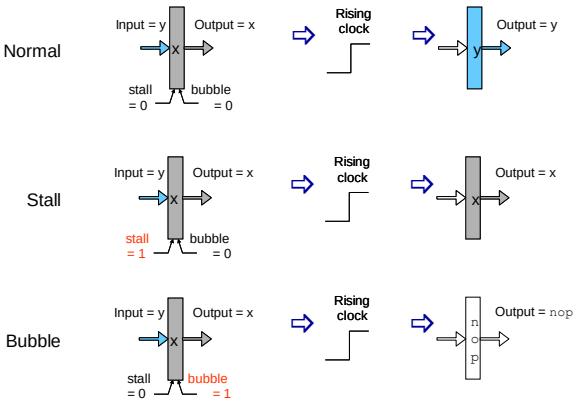


- Using a "**scoreboard**". Each register has a bit.
- Every instruction that writes to a register sets the bit.
- Every instruction that reads a register would have to check the bit first.
  - If the bit is set, then generate a bubble
  - Otherwise, free to go!!

Stalling X3



How are Stall and Bubble Implemented in Hardware?



Data Forwarding

Naïve Pipeline
- Register isn't written until completion of write-back stage
- Source operands read from register file in decode stage
- The decode stage can't start until the write-back stage finishes

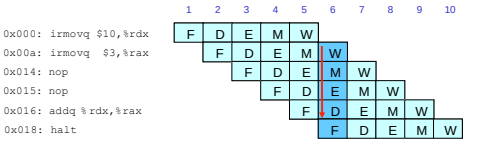
Observation
- Value generated in execute or memory stage

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

138

138

Data Forwarding Example



- `irmovq` writes `%rax` to the register file at the end of the write-back stage
- But the value of `%rax` is already available at the beginning of the write-back stage
- Forward `%rax` to the decode stage of `addq`.

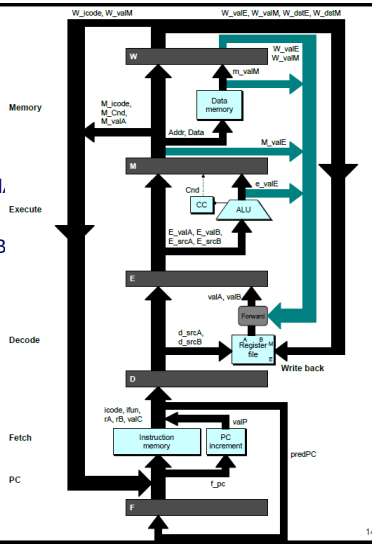
139

139

Bypass Paths

Decode Stage
- Forwarding logic selects val
- Normally from register file
- Forwarding: get valA or valB

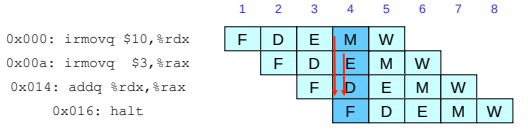
Forwarding Sources
- Execute: valE
- Memory: valE, valM
- Write back: valE, valM



140

140

Data Forwarding Example #2



Register `%rdx`
- Forward from the memory stage

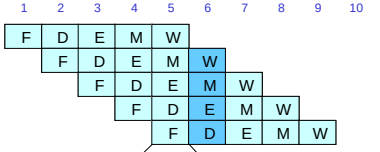
Register `%rax`
- Forward from the execute stage

141

141

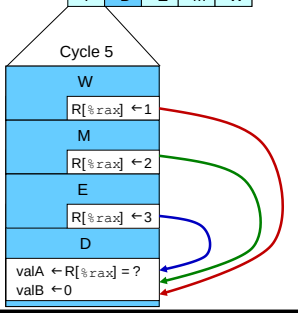
Forwarding Priority

```
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```



Multiple Forwarding Choices

- Which one should have priority
- Match serial semantics
- Use matching value from earliest pipeline stage



142

142

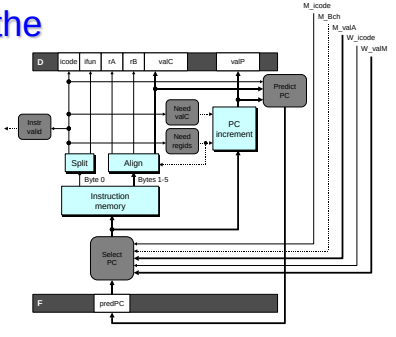
Making the Pipeline Really Work

- Data Dependencies
  - What is it?
  - Software mitigation: Inserting Nops
- Control Dependencies
  - What is it?
  - Software mitigation: Inserting Nops
  - Software mitigation: Delay Slots

143

143

Predicting the PC



- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

10/14/2020 144

144

One 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
  - Predict next PC to be valC (destination)
  - Only correct if branch is taken
    - Typically right 60% of time
- Return Instruction
  - Don't try to predict

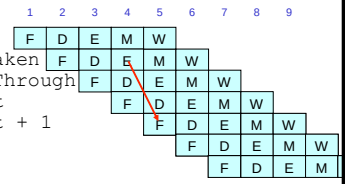
10/14/2020 145

145

Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

```
   xorg %rax, %rax
    jne L1              # Not taken
 nop irmovq $1, %rax    # Fall Through
L1nop irmovq $4, %rcx   # Target
    irmovq $3, %rax     # Target + 1
```



146

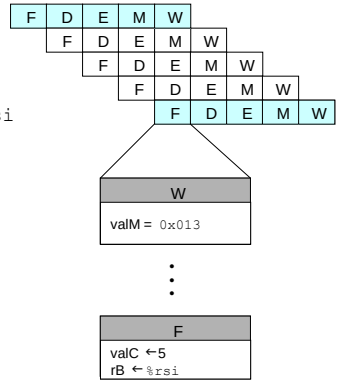
Control Dependency: Return Example

```
0x000:     irmovq Stack,%rsp  # Intialize stack pointer
0x00a:     call p             # Procedure call
0x013:     irmovq $5,%rsi     # Return point
0x01d:     halt
0x020: p:  irmovq $-1,%rdi    # procedure
0x02a:     ret
0x02b:     irmovq $1,%rax     # Should not be executed
0x035:     irmovq $2,%rcx     # Should not be executed
0x03f:     irmovq $3,%rdx     # Should not be executed
0x049:     irmovq $4,%rbx     # Should not be executed
0x100: Stack:                 # Stack: Stack pointer
```


147

Control Dependency: Correct Return

```
0x02a:   ret
         nop
         nop
         nop
0x013:   irmovq $5,%rsi
```

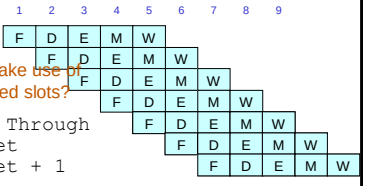


148

Delay Slots

```
   xorg %rax, %rax
   jne L1
nop
nop
   irmovq $1, %rax    # Fall Through
L1 irmovq $4, %rcx    # Target
   irmovq $3, %rax    # Target + 1
```

Can we make use of the 2 wasted slots?



Have to make sure `do_C` doesn't depend on `do_A` and `do_B`!!!

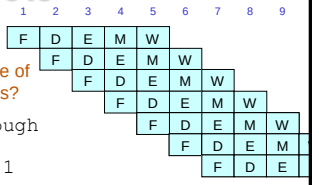
```
if (cond) {
  do_A();
} else {
  do_B();
}
do_C();
```


149

Delay Slots

```
    xorg %rax, %rax
    jne L1
    nop
    nop
    irmovq $1, %rax   # Fall Through
L1  irmovq $4, %rcx   # Target
    irmovq $3, %rax   # Target + 1
```

Can we make use of the 2 wasted slots?



Another example

```
do_C();
if (cond) {
  do_A();
} else {
  do_B();
}
```

```
add A, B
or C, D
sub E, F
jle 0x200
nop
nop
add A, C
```

```
sub E, F
jle 0x200
add A, B
or C, D
add A, C
```

Why don't we move the `sub` instruction?

150

150

Resolving Control Dependencies

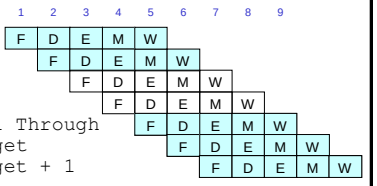
- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
  - Delay slot: insert instructions that do not depend on the effect of the preceding instruction. These instructions will execute even if the preceding branch is taken — old RISC approach
- Hardware mechanisms
  - Stalling
  - Branch Prediction
  - Return Address Stack

151

151

Hardware Stalling

```
    xorg %rax, %rax
    jne L1
  bubble
  bubble
    irmovq $1, %rax   # Fall Through
L1  irmovq $4, %rcx   # Target
    irmovq $3, %rax   # Target + 1
```



152

152