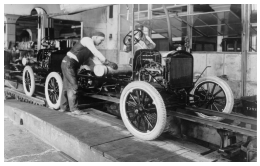
Henry Ford's Big Idea

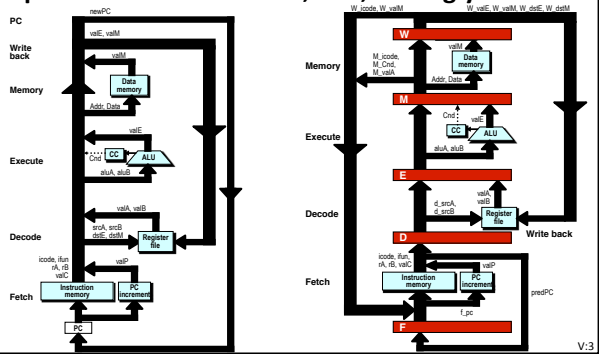


- **Assemble each car on a moving conveyor belt**
  - Cut assembly time from 13.5 hours (1913) to 93 minutes (1914)
  - Reduced cost to build, opened up mass market
    - 1908: price ≈ 18 months pay (for average US worker)
    - 1918: price ≈ 5 months pay

Pipelining

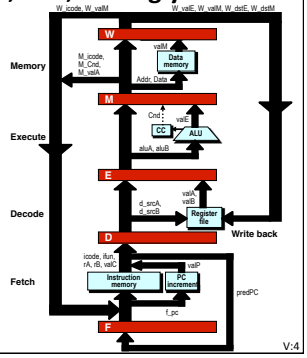
- **Key ideas**
  - Divide process into independent stages
  - Move objects through stages in sequence
  - At any instant, multiple objects are being processed
- **Impact on performance**
  - Does this reduce the processing time for a given object?
  - Does it increase the *throughput*?
- **Can you think of real-world examples?**

Pipelined CPUs: The Good, Bad, and Ugly

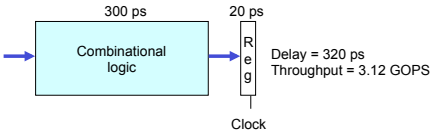


Pipelined CPUs: The Good, Bad, and Ugly

- **The big picture**
  - What speedup is expected from a 5 stage pipeline?
  - Does the pipeline make an individual instruction execute faster?
  - What problems arise when we pipeline our processor?



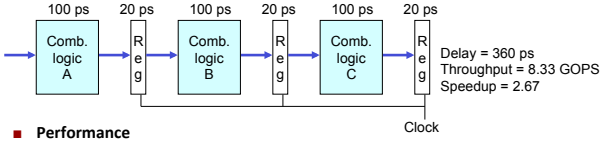
Computational Example



- **Performance**
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Clock cycle must be at least 320 ps

$$\text{Throughput} = \frac{1 \text{ operation}}{320 \text{ ps}} \times \frac{1000 \text{ ps}}{1 \text{ ns}} \approx 3.12 \text{ GOPS}$$

3-Stage Pipelined Version



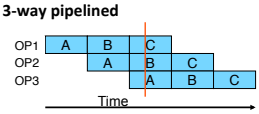
- **Performance**
  - Combinational logic divided into 3 blocks of 100 ps each
  - New operation can begin as soon as previous one passes through stage A.
    - Can start new operation every 120 ps
  - Performance impact: intuitively we'd expect about ~3x improvement
    - *Latency* increases: 360 ps from start to finish (was 320)
    - *Speedup* is less than 3x – why?

Execution Models

- **Unpipelined**

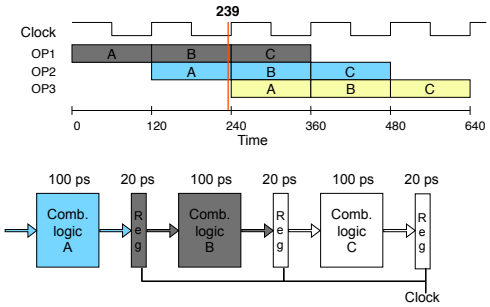
  - New operation starts when previous one finishes

- **3-way pipelined**

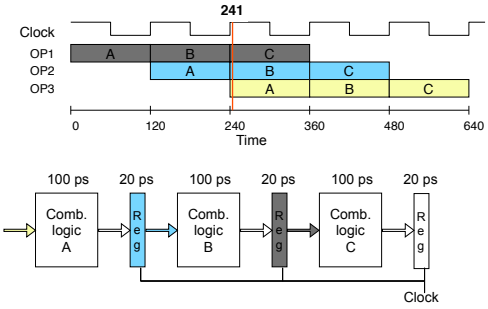


  - Up to 3 operations in process simultaneously

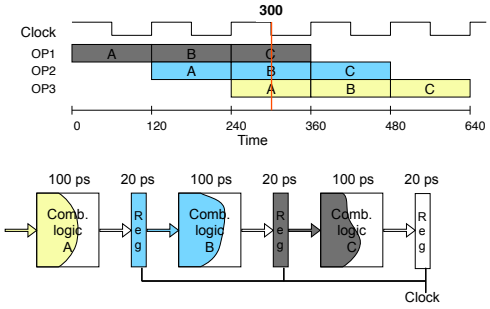
Pipeline Operation



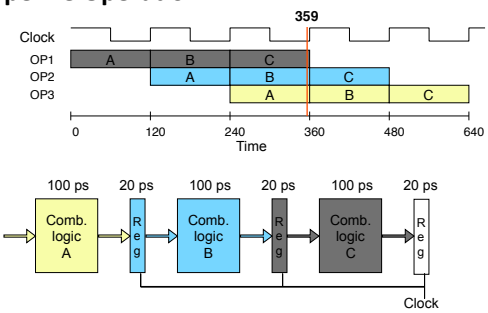
Pipeline Operation



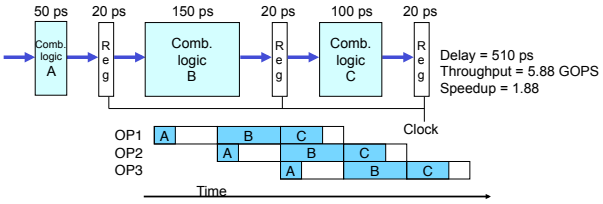
Pipeline Operation



Pipeline Operation



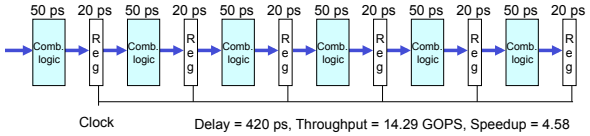
Limitations: Nonuniform Delays



Delay = 510 ps
Throughput = 5.88 GOPS
Speedup = 1.88

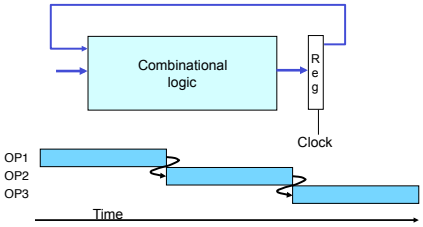
- Difficult to partition system into perfectly balanced stages
- Throughput limited by slowest stage
- Other stages sit idle for much of the time

Limitations: Register Overhead



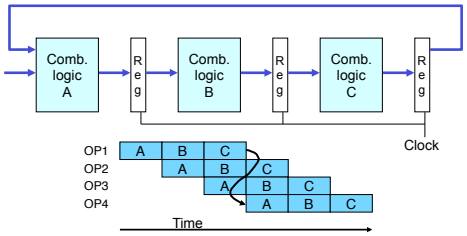
- For longer pipelines, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register (in this specific case):
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High clock rates of modern processor designs obtained through deep pipelining: comes with high overhead

Data Dependencies



- Here, each operation depends on result from preceding one
- Is this a problem in the SEQ implementation?

Limitations: Data Hazards



- Result cannot feed back around in time to start next operation
- This is a **hazard**: unless special action is taken, results will be incorrect!

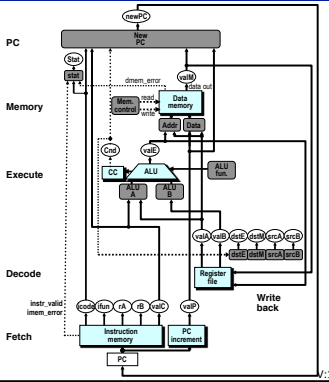
Data Dependencies in Y86-64 Code

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

- Result from one instruction is source operand for a following instruction
  - Each instance is a read-after-write (RAW) dependency
- Very common in actual programs
- Our pipeline must handle these properly
  - **Essential**: results of program execution must be correct
  - **Desirable**: minimize negative impact on performance

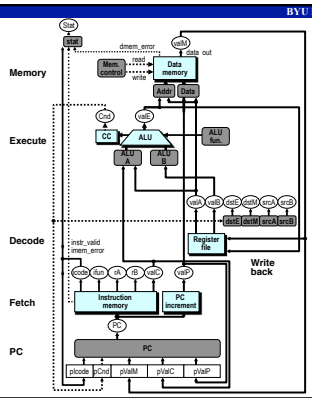
SEQ Hardware

- Stages occur in sequence
- Instructions processed one at a time

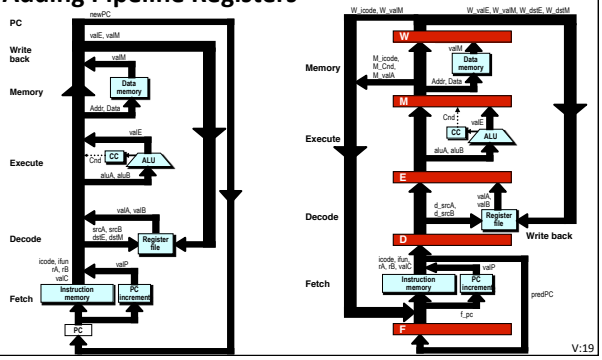


SEQ+ Hardware

- Still a sequential implementation
- PC "stage" moved to beginning
- Motivation: must have next PC by end of each cycle
- **PC selection logic**
  - Selects PC for current instruction
  - Uses values computed by previous instructions
- **Processor state**
  - PC is not a register – it is value determined from other registers
  - Determining next PC does not require separate stage

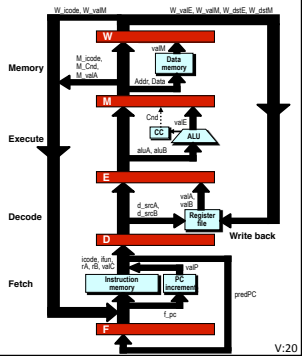


Adding Pipeline Registers



Pipeline Stages

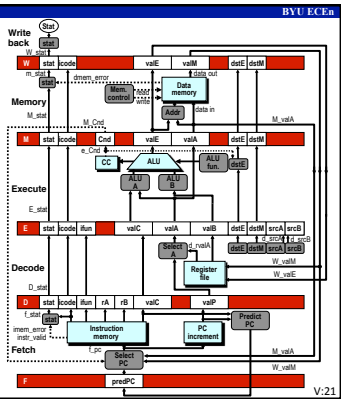
- **Fetch**
  - Select current PC
  - Read instruction
  - Compute incremented PC
- **Decode**
  - Read program registers
- **Execute**
  - Perform ALU operation
- **Memory**
  - Read or write data memory
- **Write back**
  - Update register file



PIPE- Hardware

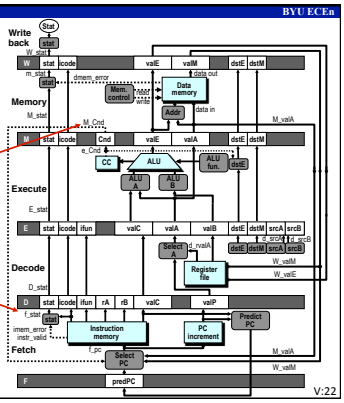
- **Pipeline registers**
  - Hold intermediate values from instruction execution
- **Forward (upward) paths**
  - Values flow from one stage to next
  - Values cannot skip stages
    - See valC in Decode, dstM in Execute, etc.

"PIPE-" name suggests that design is not yet complete



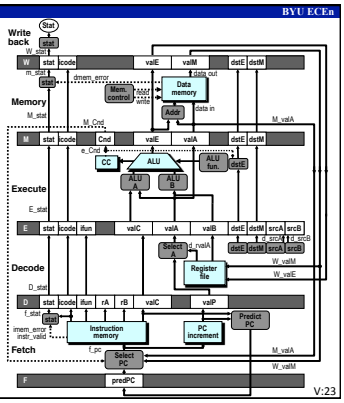
Signal Naming Conventions

- **S_Field**
  - Value of Field *stored* in stage S pipeline register
  - Example: M_Cnd
- **s_Field**
  - Value of Field *computed* in stage S
  - Example: f_stat



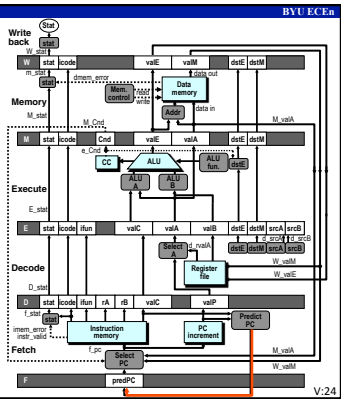
Problems: Feedback Paths

- **Predicted PC**
  - Compute or ***guess*** value of next PC
- **Branch information**
  - Jump taken/not-taken
  - Fall-through or target address
- **Return address**
  - Read from memory
- **Register updates**
  - To register file write ports



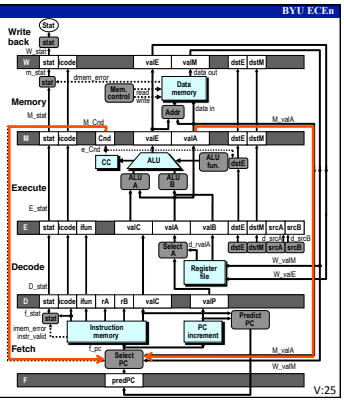
Problems: Feedback Paths

- **Predicted PC**
  - Compute or ***guess*** value of next PC
- **Branch information**
  - Jump taken/not-taken
  - Fall-through or target address
- **Return address**
  - Read from memory
- **Register updates**
  - To register file write ports



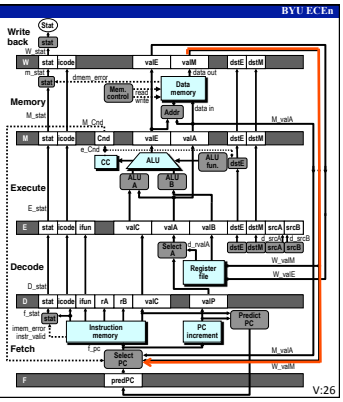
Problems: Feedback Paths

- **Predicted PC**
  - Compute or ***guess*** value of next PC
- **Branch information**
  - Jump taken/not-taken
  - Fall-through or target address
- **Return address**
  - Read from memory
- **Register updates**
  - To register file write ports



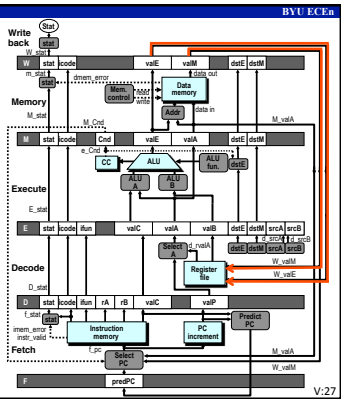
Problems: Feedback Paths

- **Predicted PC**
  - Compute or ***guess*** value of next PC
- **Branch information**
  - Jump taken/not-taken
  - Fall-through or target address
- **Return address**
  - Read from memory
- **Register updates**
  - To register file write ports



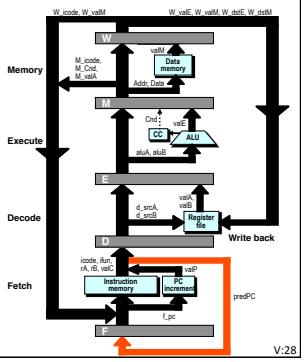
Problems: Feedback Paths

- **Predicted PC**
  - Compute or ***guess*** value of next PC
- **Branch information**
  - Jump taken/not-taken
  - Fall-through or target address
- **Return address**
  - Read from memory
- **Register updates**
  - To register file write ports



Predicting the PC

- **Pipeline timing**
  - Next instruction should be fetched one cycle after current instruction
  - Problem: not enough time to determine next instruction PC in all cases
  - Can be done for all but conditional jump and return
- **Solution for conditional jumps**
  - **Guess branch outcome** + continue!
  - If prediction is incorrect, recover later
    - Undo actions, get back on track



A Simple Prediction Strategy

- **Remember: for most instructions, next PC is known in Fetch**
  - Instructions that don't change control flow
    - Next PC is always valP (no guess required)
  - Some that change control flow: calls, unconditional jumps
    - Next PC is always valC (no guess required)
- **Only time we guess: conditional jumps**
  - Could predict
    - valC as next PC; correct if branch is taken (~60% of time)
    - valP as next PC; correct if branch not taken (~40% of time)
    - Taken if backward, not-taken if forward (correct ~65% of time)
- **Special case: return instruction**
  - Don't try to predict (Why? What would be required?)

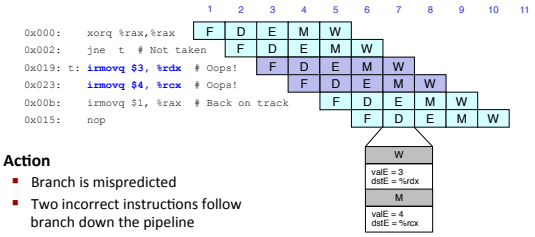
Chosen for our implementation

Branch Misprediction Example

```
0x000:     xorq %rax,%rax
0x002:     jne  t              # Not taken
0x00b:     irmovq $1, %rax     # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx     # Target (Should not execute)
0x023:     irmovq $4, %rcx     # Should not execute
0x02d:     irmovq $5, %rdx     # Should not execute
```

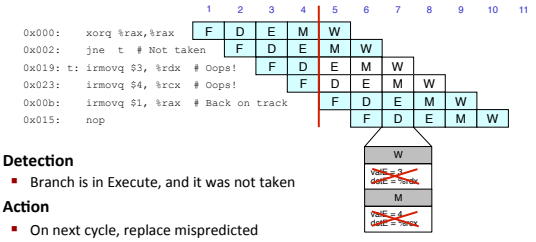
- Should execute only first 7 instructions
- Our predictor will (incorrectly) guess that the branch will be taken
- What will happen then, and how can pipeline recover?

Handling Misprediction: Incorrect Behavior



- **Action**
  - Branch is mispredicted
  - Two incorrect instructions follow branch down the pipeline

Handling Misprediction: Desired Behavior



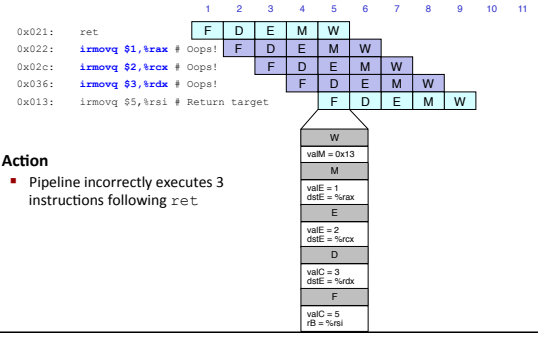
- **Detection**
  - Branch is in Execute, and it was not taken
- **Action**
  - On next cycle, replace mispredicted instructions with "bubbles"
    - Dynamically inserted nops

Return Example

```
0x000:     irmovq Stack,%rsp    # Intialize stack pointer
0x00a:     call p               # Procedure call
0x013:     irmovq $5,%rsi       # Return point
0x01d:     halt
           .pos 0x20
0x020: p:  nop                  # procedure
0x021:     ret
0x022:     irmovq $1,%rax       # Should not be executed
0x02c:     irmovq $2,%rcx       # Should not be executed
0x036:     irmovq $3,%rdx       # Should not be executed
0x040:     irmovq $4,%rbx       # Should not be executed
           .pos 0x100
0x100: Stack:                   # Initial stack pointer
```

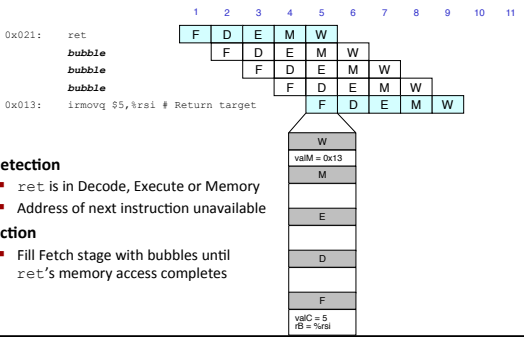
- Without special handling, machine would fetch three more instructions after `ret` before return address is known
- Our approach: freeze the pipe when `ret` is fetched
- Why not try to predict?

Handling Return: Incorrect Behavior



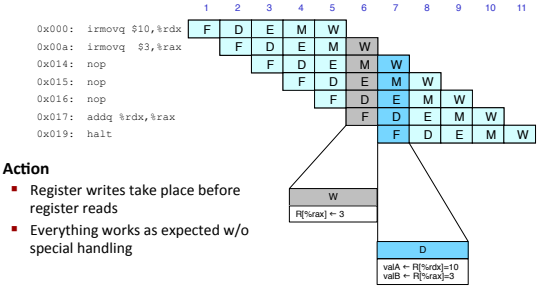
- **Action**
  - Pipeline incorrectly executes 3 instructions following `ret`

Handling Return: Desired Behavior



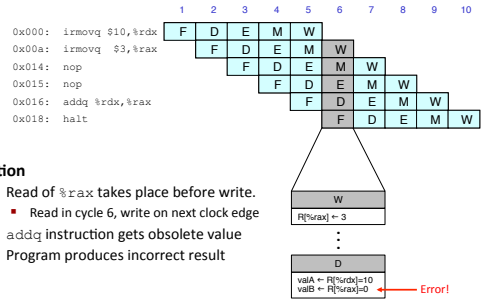
- **Detection**
  - `ret` is in Decode, Execute or Memory
  - Address of next instruction unavailable
- **Action**
  - Fill Fetch stage with bubbles until `ret`'s memory access completes

Data Dependencies: 3-Nop Separation



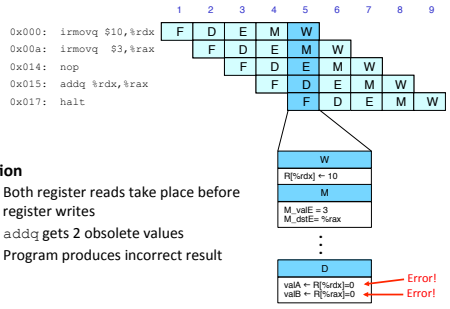
- **Action**
  - Register writes take place before register reads
  - Everything works as expected w/o special handling

Data Dependencies: 2-Nop Separation



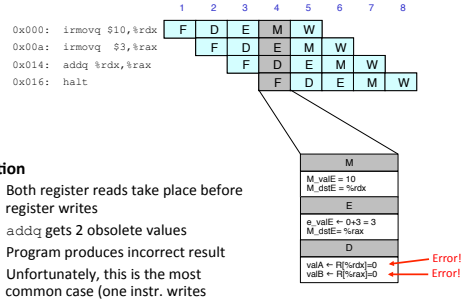
- **Action**
  - Read of `%rax` takes place before write.
    - Read in cycle 6, write on next clock edge
  - `addq` instruction gets obsolete value
  - Program produces incorrect result

Data Dependencies: 1-Nop Separation



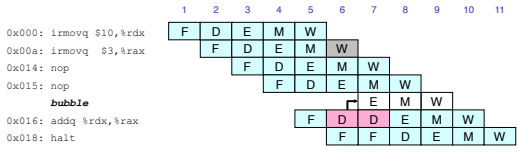
- **Action**
  - Both register reads take place before register writes
  - `addq` gets 2 obsolete values
  - Program produces incorrect result

Data Dependencies: 0-Nop Separation



- **Action**
  - Both register reads take place before register writes
  - `addq` gets 2 obsolete values
  - Program produces incorrect result
  - Unfortunately, this is the most common case (one instr. writes register, next instr. reads it)

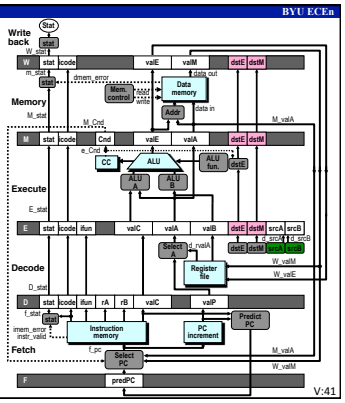
Hazard Solution 1: Stalling



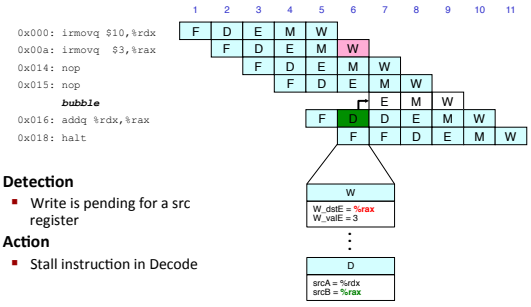
- If instruction that reads register follows too closely after instruction that writes same register, delay the reading instruction
- Stall (delay) always takes place in Decode stage
  - Following instruction also delayed (in Fetch), but just one "stall cycle" tallied
- Stall = dynamically injecting nop (bubble) into execute stage
- **Hazard ⇒ we get wrong answer without special action**

Stalls

- **Detection**
  - Write is pending for a src register
  - Therefore, srcA or srcB matches dstE or dstM in Execute, Memory, or Write-back stages
- **Action**
  - Stall instruction in Decode
  - No action if reg ID is 0xF

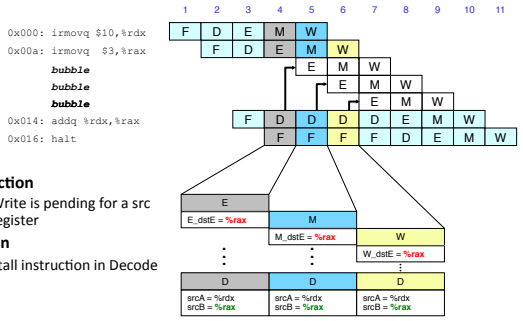


Detecting Stall Condition



- **Detection**
  - Write is pending for a src register
- **Action**
  - Stall instruction in Decode

Multi-cycle Stalls



- **Detection**
  - Write is pending for a src register
- **Action**
  - Stall instruction in Decode

Stalling: Another View

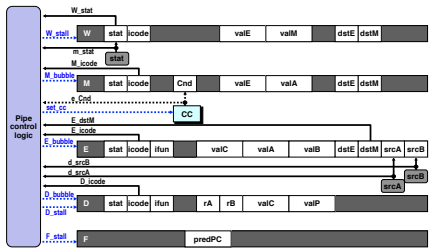
```
0x000:  irmovq  $10,%rdx
0x00a:  irmovq  $3,%rax
0x014:  addq %rdx,%rax
0x016:  halt
```

Cycle 8

Write Back	***bubble***
Memory	***bubble***
Execute	`0x014: addq %rdx,%rax`
Decode	`0x016: halt`
Fetch	

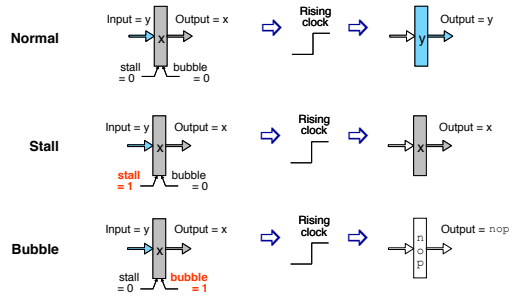
- **Operational rules:**
  - Stalling instruction stays in Decode stage
  - Following instruction stays in Fetch stage
  - Bubbles injected into Execute stage
    - Like dynamically generated nops
    - Move through stage-by-stage as regular instructions

Implementing Stalling



- **Pipeline control**
  - Combinational logic detects stall condition from inputs
  - Asserts Register Mode signals that control pipeline register operation

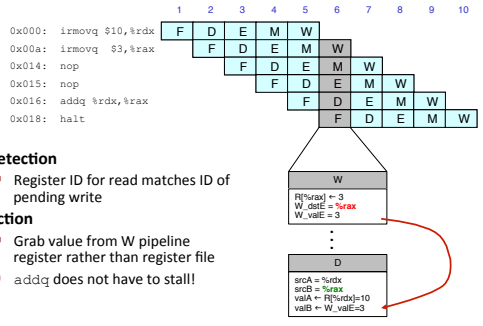
Pipeline Control: Register Modes



Hazard Solution 2: Data Forwarding

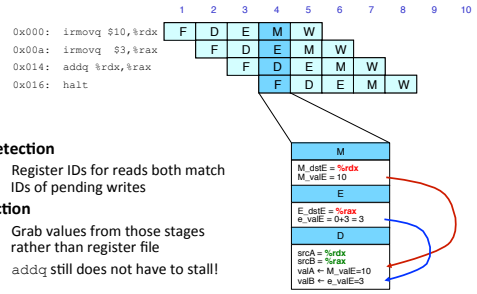
- **Our naïve pipeline would experience *many* data stalls**
  - Result register isn't written until completion of Write-back stage
  - Source operands are read from register file in Decode stage; values need to be in register file at *start* of Decode
- **Key observation**
  - The value we want is generated in Execute or Memory stages
  - It exists 1-2 cycles before it is written to register file
- **Basic idea of forwarding: *go get the value you want!***
  - Pass value directly from its stage to Decode stage
  - If available before end of Decode, we can avoid stall

Data Forwarding Example #1



- **Detection**
  - Register ID for read matches ID of pending write
- **Action**
  - Grab value from W pipeline register rather than register file
  - `addq` does not have to stall!

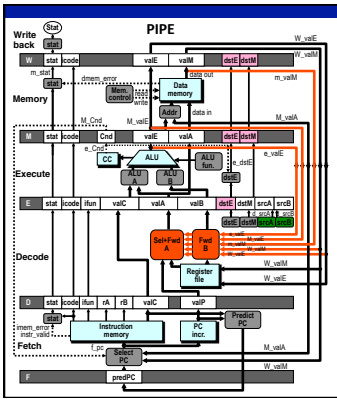
Data Forwarding Example #2



- **Detection**
  - Register IDs for reads both match IDs of pending writes
- **Action**
  - Grab values from those stages rather than register file
  - `addq` still does not have to stall!

So, how does one "grab values" like this?

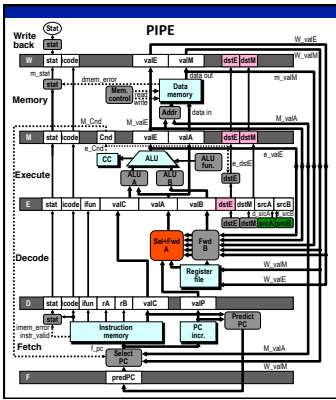
PIPE Forwarding Hardware



- Feedback paths from E, M, and W registers to Decode stage
- Logic blocks to select source for valA and valB in Decode stage
- Note: for data hazards, we must either do forwarding or stall
  - Forwarding has much better performance, but higher cost

Note name: "PIPE" is final, complete version

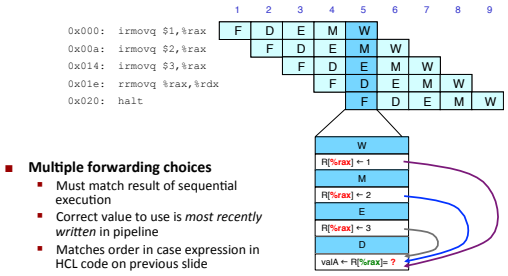
PIPE Forwarding Control




```
## Actions of "Sel+Fwd A" block
## Pick the correct A value
## Order is important!
int new_E_valA = [
  # Use incremented PC
      D_icode in {ICALL,IJXX} : D_valP;
  # Forward valE from Execute
      d_srcA == E_dstE : e_valE;
  # Forward valM from Memory
      d_srcA == M_dstM : m_valM;
  # Forward valE from Memory
      d_srcA == M_dstE : M_valE;
  # Forward valM from Write back
      d_srcA == W_dstM : W_valM;
  # Forward valE from Write back
      d_srcA == W_dstE : W_valE;
  # Use value read from register file
      1 : d_rvalA;
];
```

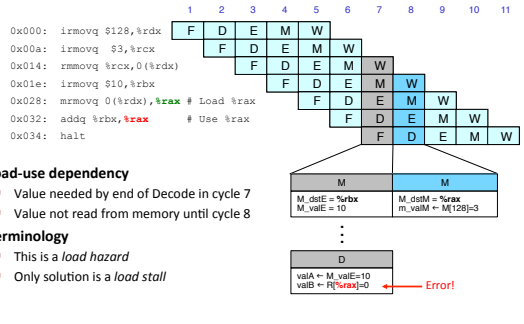
Why is order important?

Forwarding Priority



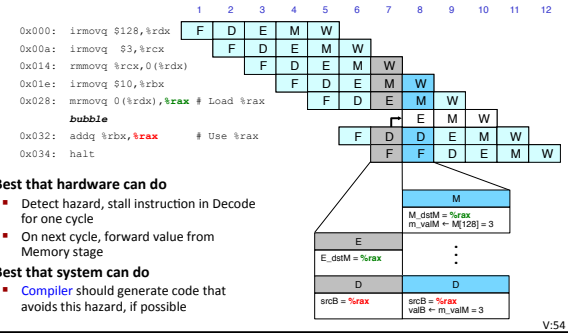
- **Multiple forwarding choices**
  - Must match result of sequential execution
  - Correct value to use is *most recently written* in pipeline
  - Matches order in case expression in HCL code on previous slide

Limitations of Forwarding



- **Load-use dependency**
  - Value needed by end of Decode in cycle 7
  - Value not read from memory until cycle 8
- **Terminology**
  - This is a *load hazard*
  - Only solution is a *load stall*

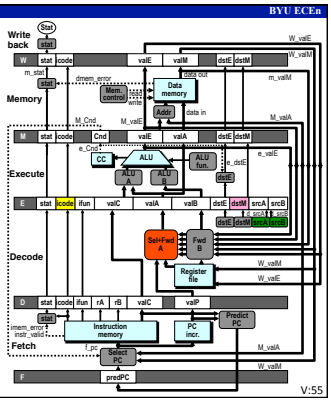
Load/Use Hazard: Desired Behavior



- **Best that hardware can do**
  - Detect hazard, stall instruction in Decode for one cycle
  - On next cycle, forward value from Memory stage
- **Best that system can do**
  - Compiler should generate code that avoids this hazard, if possible

Handling Load/Use Hazard

- **Detection**
  - Instr. in Execute will load value from memory to register X
  - Instr. in Decode is trying to read register X
  - HW details: E_dstM (not 0xF) matches d_srcA or d_srcB
- **Action**
  - Stall instruction in Decode

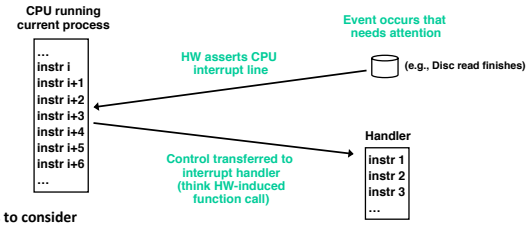


Standard for "Correct" Execution

- **Results must match those of sequential implementation**
  - Challenges we had to address in pipeline
    - Branch mispredictions
    - `ret` instruction
    - Data hazards
  - Convince yourself that PIPE design results in correct execution
- **Additional challenge, less obvious than data and control flow**
  - High performance CPU must match exception behavior of sequential CPU
  - Should experience exception if and only if it occurred in sequential machine
  - Harder to get this behavior than you might think!
  - Let's explore...

Interrupts and Exceptions

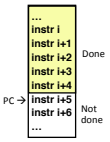
- **Review: basic interrupt mechanism**



- **Details to consider**
  - What state of current process is saved?
  - How/where is it saved?
  - How is location of handler determined?

Interrupt Handling

- **Hardware saves some state and causes handler to run**
  - State saved on stack, includes single PC value
    - Address of instruction to return to (current or next instruction)
    - Usually passes through pipeline with instruction
    - **Precise exception**: all instructions to PC executed, none past
  - Transfer of control to handler – part of OS
    - **Interrupt vector table** stored in memory at fixed address
    - Exception/interrupt number used as index; table contains handler addresses
    - Table contents written by software, accessed by hardware



- **Implementation**
  - Critical for real hardware
  - Seldom implemented in simulators: no OS running to pass control to!
    - Exception/interrupt handling in Y86 is not realistic

Exceptions

Events that occur *within* processor under which pipeline cannot continue normal operation

- **Possible causes**
  - Halt instruction executed (Current)
  - Bad address for instruction or data (Previous)
  - Invalid instruction (Previous)
  - Pipeline control error (Previous)
  - System calls, page faults, math errors (not in Y86-64)
- **Desired actions (of hardware)**
  - Make exception **precise**
    - Complete all instructions to specific point (current or previous)
    - Discard any and all (partially executed) instructions that follow
  - Transfer control to exception handler in OS
    - Save return address (PC), get handler address from table

Exception Examples: Y86-64

- **Detect in fetch stage**

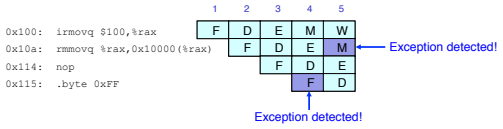
```
jmp $-1                 # Invalid jump target address

.byte 0xFF              # Invalid instruction code

halt                    # Halt instruction
```

- **Detect in memory stage**

```
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # Invalid address (for Y86-64 tools)
```

Handling Exceptions (#1)

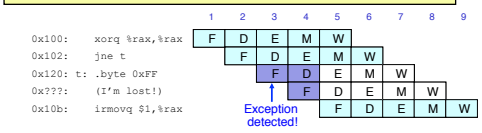
```
    irmovq $100,%rax
    rmmovq %rax,0x10000(%rax)     # Invalid address
    nop
    .byte 0xFF                    # Invalid instruction code
```



- **Desired behavior**
  - Must match behavior of sequential CPU: respond to exception on `rmmovq`
  - Tricky because another exception was detected first

Handling Exceptions (#2)

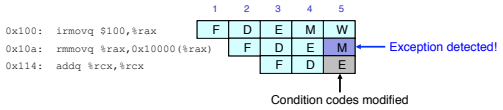
```
0x100:    xorq %rax,%rax      # Set condition codes
0x102:    jne t               # Not taken
0x10b:    irmovq $1,%rax
0x115:    irmovq $2,%rdx
0x11f:    halt
0x120: t: .byte 0xFF          # Target
```



- **Desired behavior**
  - Must match behavior of sequential CPU: no exception should "occur"
  - Tricky because an exception is detected

Handling Exceptions (#3)

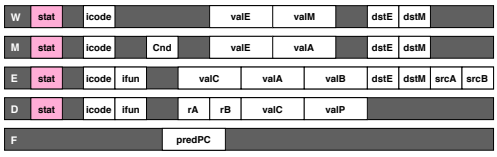
```
    irmovq $100,%rax
    rmmovq %rbx,0x10000(%rax)   # Invalid address
    addq   %rcx,%rcx            # Sets condition codes
```



- **Desired behavior**
  - `rmmovl` should cause exception
  - `addq` instruction must not be allowed to change state
    - Condition codes are a special challenge! (Worse than program registers)

Correct Exception Handling

- **Challenge: respond to exceptions in program order, and only "real" ones**
- **Solution**
  - Add stat field to each pipeline register: record exceptions for instruction in that stage
  - Fetch stage sets to "AOK," "ADR" (bad address), "INS" (illegal instruction), or "HLT"
  - Decode & execute stages pass values through
  - Memory stage either passes through or sets to "ADR"
  - CPU responds to exception only when instruction reaches write back (Why?)

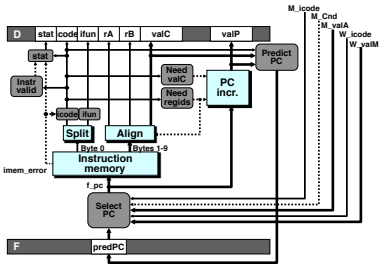


Avoiding Side Effects

- **Exception must disable all state updates for instructions that follow**
  - If exception detected in Memory stage:
    - Disable condition code setting in Execute (in same cycle!)
  - When exception passes to Write-back stage
    - Disable write to memory in Memory stage
    - Disable condition code setting in Execute stage
- **We'll see shortly how these are handled in PIPE processor**

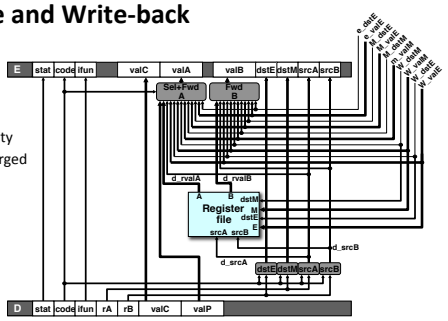
PIPE: Fetch Details

- **Main points**
  - Branch prediction
  - Branch misprediction recovery
  - Return handling
  - Stat initialization



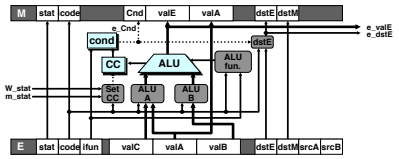
PIPE: Decode and Write-back

- **Main points**
  - Forwarding logic, paths
  - Forwarding priority
  - valA and valP merged



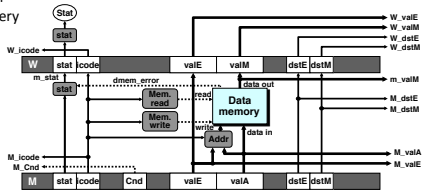
PIPE: Execute

- **Main points**
  - CC update inhibited by prior exceptions
  - Values for forwarding
  - Logic block for dstE supports conditional moves

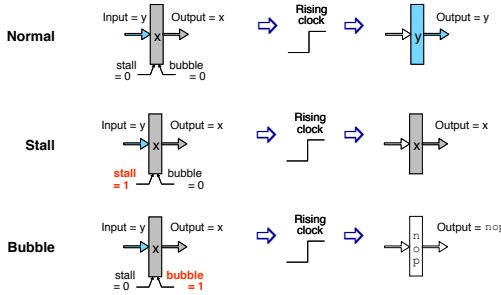


PIPE: Memory and Write-back

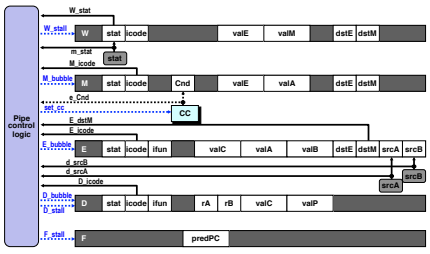
- **Main points**
  - Values for forwarding
  - Stat update logic
  - Feedback for branch misprediction recovery



Pipeline Control: Register Modes

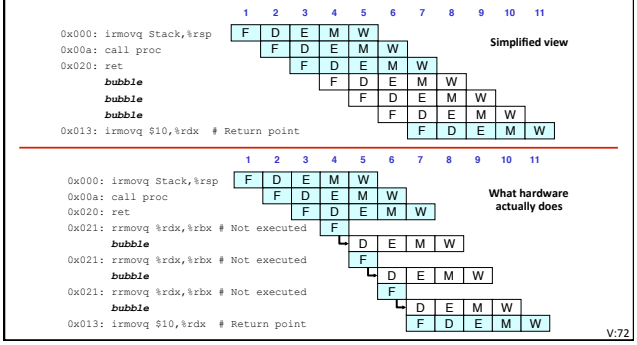


PIPE Control Logic



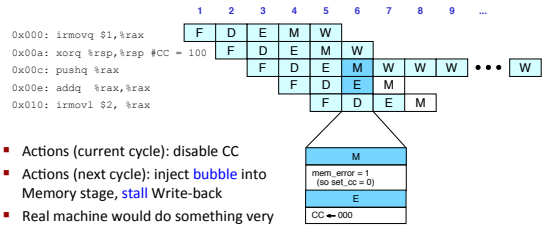
- **Handles special cases**
  - Handles ret, load/use hazards, misprediction recovery, exceptions
  - Existing PIPE logic handles forwarding, branch prediction

PIPE: Actual `ret` handling



PIPE: Actual exception handling

- **Scenario: pushq uses bad memory address**



- Actions (current cycle): disable CC
- Actions (next cycle): inject bubble into Memory stage, stall Write-back
- Real machine would do something very different (cause handler to run)

Special Control Cases: Exceptions

- **Detection**

Condition	Trigger
Exception	m_stat in {SADR, SINS, SHLT} II W_stat in {SADR, SINS, SHLT}

- **Action (on next cycle)**

Condition	F	D	E	M	W
Exception	normal	normal	normal	bubble	stall

- Also: disable setting of condition codes (in Execute) in current cycle
- Effect: no following instructions can change state, excepting instructions stays in Write-back (exception code stays in Stat register)

Special Control Cases: Non-exceptions

- **Detection**

Condition	Trigger
Processing `ret`	IRET in { D_icode, E_icode, M_icode }
Load/use hazard	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
Mispredicted branch	E_icode = IJXX & !e_Cnd

- **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

Pipeline Control, ver. 1.0

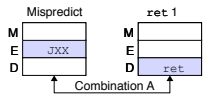
```
bool F_stall =
      # Conditions for a load/use hazard
      E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB } ||
      # Stalling at fetch while ret passes through pipeline
          IRET in { D_icode, E_icode, M_icode };

bool D_stall =
      # Conditions for a load/use hazard
      E_icode in { IMRMOVQ, IPOPQ } && 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) ||
      # Conditions for a load/use hazard
          E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB};
```

How do we know this works?

Verification

- **What could designers do to make sure pipeline control really works?**
- **Extensive simulation?**
  - Simulation can show that errors exist, but not that design is error free.
- **Careful analysis?**
  - Maybe special cases *in combination* aren't handled correctly.
  - What combinations might exist of these?
    - Mispredicted branch
    - Handling ret
    - Load/use hazard

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 be handled as mispredicted branch
- Combination will also stall F pipeline register
- But PC selection logic will be using M_valA anyway
- Correct action is taken!

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 assert both bubble *and* stall for pipeline register D
  - In authors' design, simulator signaled as pipeline error (exception)
- Combination not handled correctly in control code ver. 1.0
  - But it passed many simulation tests; caught only with systematic analysis

Control Combination B: Correct Handling

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Combination	*stall*	*stall*	*bubble*	*normal*	*normal*

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

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 { IMRMOVQ, IPOPQ }
           && E_dstM in { d_srcA, d_srcB });
```

New

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Combination	*stall*	*stall*	*bubble*	*normal*	*normal*

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

Lesson Learned

- **Extensive and thorough testing is good**
  - But it can't prove a design correct
- **Formal verification would be very valuable**
  - But field not yet mature enough for large-scale designs
  - Important and active research area

Performance Metrics

- **Clock rate (MHz or GHz)**
  - Function of stage partitioning and circuit design
  - Can increase by *reducing* work done per stage
- **Rate at which instructions finish**
  - CPI: cycles per instruction
    - Clock cycles required (on average) to complete current instruction *after* completion of previous instruction.
  - CPI a function of both pipeline design and program
    - PIPE loses one cycle for each bubble that is inserted
    - How frequently do bubbles occur?

CPI for PIPE

- **Ideal CPI = 1.0**
  - Achieved if one useful instruction is completing each clock cycle
  - Very different from latency (instruction takes ≥ 5 cycles to get through pipe)
- **Actual CPI > 1.0**
  - Due to load stalls, branch mispredictions, ret instructions
- **Computing the CPI**
  - C clock cycles
  - I instructions executed to completion
  - B bubbles injected
    C = I + B
    CPI = C/I = (I+B)/I = 1.0 + B/I
  - B/I represents average penalty (per instruction) due to bubbles

CPI for PIPE (Cont.)

Average penalty **B/I = LP + MP + RP**

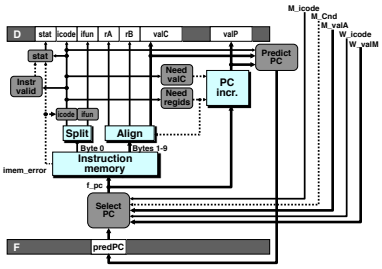
- **LP**: Penalty due to load/use hazard stalling — Typical Values
  - Fraction of instructions that are loads — 0.25
  - Fraction of load instructions requiring stall — 0.20
  - Number of bubbles injected each time — 1
  - ⇒ **LP = 0.25 * 0.20 * 1 = 0.05**
- **MP**: Penalty due to mispredicted branches
  - Fraction of instructions that are cond. jumps — 0.20
  - Fraction of cond. jumps mispredicted — 0.40
  - Number of bubbles injected each time — 2
  - ⇒ **MP = 0.20 * 0.40 * 2 = 0.16**
- **RP**: Penalty due to `ret` instructions
  - Fraction of instructions that are returns — 0.02
  - Number of bubbles injected each time — 3
  - ⇒ **RP = 0.02 * 3 = 0.06**
- Net effect of penalties: 0.05 + 0.16 + 0.06 = 0.27
  - ⇒ **CPI = 1.27** (Not bad! Gets worse with more realistic memories.)

State-of-the-Art Pipelining

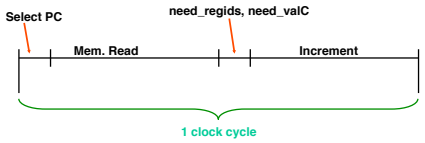
- **What have we ignored in our Y86 implementation?**
  - Balancing delay in each stage
  - Which stage is longest?
    - Fetch, Decode, Execute, Memory, or Write-back?
  - How might we speed up that slowest stage?

Fetch Logic Revisited

- **During Fetch cycle**
  1. Select PC
  2. Read bytes from instruction memory
  3. Examine icode to determine instr. length
  4. Increment PC
- **Timing**
  - Steps 2 & 4 are typically slow

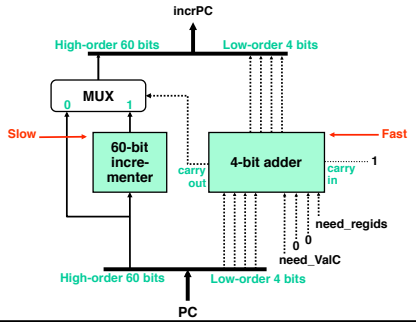


Standard Fetch Timing

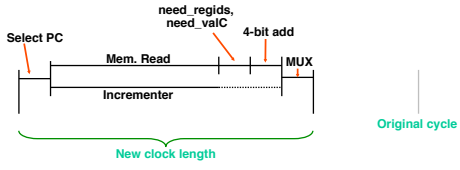


- Steps must be performed in sequence:
  - Can't read instruction from memory until PC determined, etc.
- Why is increment slow?
- Is there any way we could speed this stage up?

Speedup Trick: A Fast PC Increment Circuit



Modified Fetch Timing



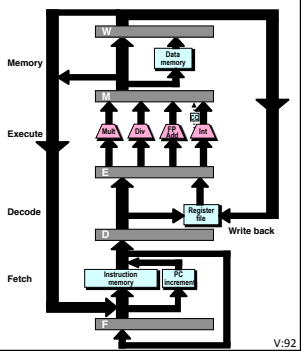
- **60-bit incrementer**
  - Acts as soon as PC selected
  - Output not needed until final MUX
  - Works in parallel with memory read

State-of-the-Art Pipelining

- **What happens when we**
  - add FP instructions?
  - extend the pipeline length?
  - add realistic memories?
  - make the CPU superscalar?
  - need better branch prediction?
  - use dynamic scheduling?

Adding FP Instructions

- **Datapath changes:**
  - Common approach is to add specialized ALUs (F-units)
- **Example: 4 F-units**
  - Multiplier (both int, FP)
  - Divider (both int, FP)
  - FP adder
  - Integer ALU (executes all other instructions)
- **How does this affect the operation of the processor?**



Timings for New ALUs

- **Virtually impossible to complete all operations in one execution cycle**
  - Instruction timings might be as shown below:

`addq`	F	D	E	M	W										
`addd`	F	D	E	E	E	M	W								
`multd`	F	D	E	E	E	E	E	M	W						
`divd`	F	D	E	E	E	E	E	E	E	E	E	E	M	W	

- **Operational details (values are arbitrary)**
  - Integer add spends one cycle in Execute, so *latency* of integer ALU is 1 cycle
    - Latency of FP adder is 3 cycles
    - Latency of multiply unit is 5 cycles
    - Latency of divide unit 11 cycles

Implications

- **Instructions can finish out of program order**
  - Although they started down pipeline in program order
  - Example:
    - `%f1`, `%f2` are FP registers
    - `cvti2d` converts a 32-bit int to a 64-bit double

`multd %f2,%f1`	F	D	E	E	E	E	E	M	W
`cvti2d %eax,%f1`		F	D	E	E	E	M	W	

  - What problem do you see in this case?

Implications

`multd %f2,%f1`	F	D	E	E	E	E	E	M	W
`cvti2d %eax,%f1`		F	D	E	E	E	M	W	

- **This instruction sequence would leave the wrong value in `%f1`**
  - This is an instance of a **WAW hazard** (write after write)
  - If we don't take special action, we will get the wrong answer
- **What can be done to ensure correct operation?**
  - Pipe control logic could test for WAW hazard in Decode, stall if necessary
  - Hardware would need to know when values will be written
  - Remember: PIPE logic already tests for **RAW hazard** in Decode

`multd %f2,%f1`	F	D	E	E	E	E	E	M	W
`cvti2d %eax,%f1`		F	D	D	D	E	E	E	M	W

Data Hazards: Are There Others?

- **Consider all possibilities with instructions *i* and *j***
  - *i* comes before *j* in program order: no hazards if far apart
  - What relationships can exist between data used by *i* and *j*?
- **RAR (read after read)**
  - *i* and *j* both read the same value
  - Never a problem – not a hazard
- **RAW (read after write)**
  - *j* reads a value that *i* writes: data dependence
  - Problem solved by forwarding or stalling
- **WAW (write after write)**
  - *i* and *j* both write the same value: output dependence
  - A problem if *j* completes before *i*
- **WAR (write after read)**
  - *i* reads a value that *j* writes: anti-dependence
  - A problem if *j* executes before *i*

```
...
instr i
...
instr j
...
```

These limit ability of compiler or hardware to reorder instructions or operations.

Variation #1

```
multd %f2,%f1
cvti2d %eax,%f3
```

F	D	E	E	E	E	E	M	W
	F	D	E	E	E	M	W	

- **Is this instruction sequence okay?**
  - The instructions are *independent* – they use different register values
  - The registers would not be updated in program order – is that a problem?

Variation #2

```
# %f1 is 0
divd %f2,%f1
addq %rax,%rbx
```

F	D	E	E	E	E	E	E	E	E	E	E	E	M	W
	F	D	E	M	W									

- **Is this instruction sequence okay?**
  - The instructions are *independent*
  - The **`divd`** will cause a divide by zero exception
  - What is the challenge here in making the exception precise?
    - When CPU responds to exception (**`divd`** reaches end of pipe), **`%rbx`** and CC will already be modified
    - Challenge: how can we prevent these modifications or undo them?

Variation #3

```
multd %f2,%f1
addd %f3,%f1
```

F	D	E	E	E	E	E	M	W		
	F	D	D	D	D	D	E	E	E	M	W

- **What will happen here?**
  - Is this a hazard? Yes.
  - If so, what kind? RAW.
  - What will execution timing look like?

Discussion

- **Forwarding and FP instructions**
  - Forwarding helped eliminate most stalls on integer instructions
  - How much does it help with FP instructions?
  - Does this example (from previous slide) assume forwarding?

```
multd %f2,%f1
addd %f3,%f1
```

F	D	E	E	E	E	E	M	W			
	F	D	D	D	D	D	E	E	E	M	W

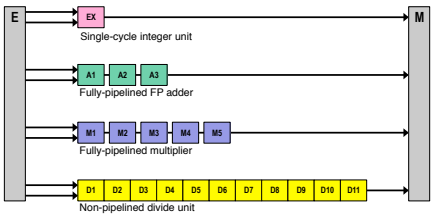
  - What would timing look like without forwarding?

```
multd %f2,%f1
addd %f3,%f1
```

F	D	E	E	E	E	E	M	W					
	F	D	D	D	D	D	D	D	D	E	E	E	M	W

F-unit Characteristics

- **Fully pipelined: a new operation may begin every cycle**
  - If not fully pipelined, start of new operation is constrained
  - For divider below, new op. must wait for *n* cycles after previous started
    - What are implications if *n* is 3, 5, or 11, say?



Hazards

- **A new type of hazard arises in F-units that are not fully pipelined**
  - **Structural hazard**: required hardware is not yet available
  - Solution: stall the instruction in Decode until the hazard clears
    - Until the required hardware becomes available
- **We're already seen:**
  - **Data hazard**: Data value is not yet available
  - **Control hazard**: Next PC value is not yet available

Quiz

- **Fundamental concepts in pipelining:**
  1. Does every data dependence cause a data hazard?
  2. Does every control dependence cause a control hazard?
  3. Is every data hazard caused by a data dependence?
  4. Does every data hazard cause a stall?
  5. Does every control hazard cause a stall?
  6. Is every stall caused by a data hazard?
  7. Can hazards occur on accesses to the condition codes?
  8. Can hazards occur on loads and stores?
  9. What can designer do to avoid structural hazards?
  10. What can designer do to avoid control hazards?

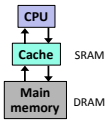
Longer Pipes: Matching Memory Speed

- **Considerable difference between CPU and memory cycle times**
  - Even with a cache, tough to access memory in single cycle
  - One option: allow multiple cycles in pipeline for each memory access
- **Suppose it takes two CPU cycles to access memory (cache)**
  - Our five-stage pipe becomes a seven-stage pipe:

- **What are consequences?**
  - Is throughput reduced? (Can we still fetch a new instruction each cycle?)
  - What is penalty of a mispredicted branch?
  - Can we still use forwarding to avoid stalls on data hazards?
  - Does this affect the penalty or frequency of load stalls?

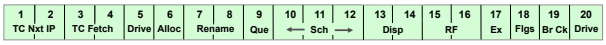
Realistic Memories

- **Spending just two cycles per memory access is optimistic**
  - Every memory reference is serviced by a memory hierarchy
    - Combines small/fast and large/slow memories
    - Access time of main memory ~50x that of cache
    - Desired: a *large* memory that is also *fast*, on average
  - Caches and hierarchy managed entirely by hardware
    - Copies of memory blocks placed in cache when accessed by CPU
- **Consequence: time to access memory varies widely**
  - Best case: cache hit, 1-3 cycles
  - Worse case: cache miss, likely 50-100+ cycles
    - Can be reduced by adding additional levels of cache
  - Realistic memory would cause loss of many more cycles in PIPE



Longer Pipes: Example

- **Pentium 4**
  - Pipeline below is for simple integer instructions – longer for FP operations
  - Results in **20+ cycle** branch misprediction penalty
  - Actually slower than predecessor (Pentium III) for a given clock rate



- **What factors motivate the design of such a deep pipeline?**
  - What work is done in each stage?

"Fat" Pipes

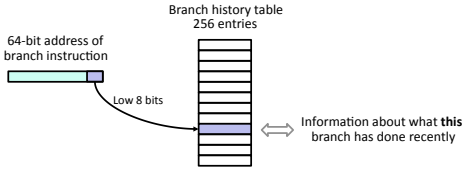
- **Could we fetch, decode, and execute *multiple* instructions per cycle?**
  - This is fundamental idea behind **superscalar** CPUs
  - Example: 4-way superscalar has potential to process 4 instructions/cycle
  - Increases utilization of ALUs already present in CPU
- **What major challenges would have to be addressed?**
  - How many bytes of memory must be fetched for the instructions?
  - What about control dependencies within the fetch block?
  - What about data dependencies within the fetch block?
- **These are hard problems to address with really fast circuitry**
  - Designers now emphasize **multi-core** rather than superscalar
  - Intel now sells multiple *moderately aggressive* CPUs on a chip, rather than a single, *very aggressive* CPU that can run a single program very fast

Branch Prediction

- **Absolutely critical to keep pipeline going**
  - Need predicted PC (for next cycle) by end of current cycle
  - Always guessing "taken" is not nearly good enough
    - Would mispredict ~40% of conditional branches
- **How can we do better?**
  - Track behavior of each branch dynamically
    - Predict based on recent behavior of that specific branch
  - Problem: only information we have (during Fetch) is *address* of instruction
  - What can we do in parallel with instruction fetch?

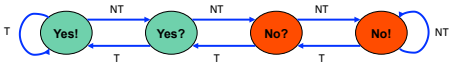
Branch History Tables

- **Select table entry using address of branch instruction**
  - Mapping is simple hash function: simple, fast in HW
    - Downside: multiple branches may map to same table entry
  - Entry reflects recent branch outcomes
  - Example:



Branch Prediction: Simplistic

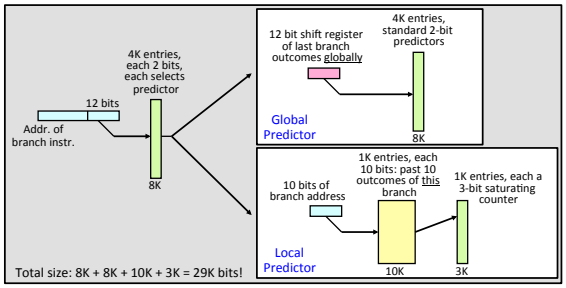
- **Branch history**
  - Here, each table entry consists of 2 bits, encoding one of 4 states.
  - Value of entry used to predict next branch outcome.



- **State machine reflects history of this branch**
  - If branch taken, move to left; else move to right.
  - In state Yes*, predict taken; in state No*, predict not taken.
- **Accuracy**
  - Will this be better than predicting all branches taken?

Branch Prediction: Complex

- **Alpha 21264 "tournament" predictor**

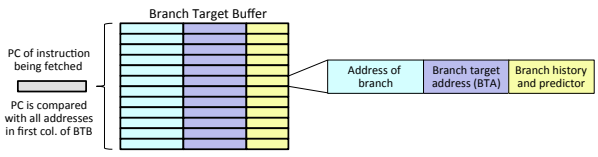


Discussion

- **Predictors on previous slides predict just the *branch outcome***
  - They do not give the branch target address
- **Would this be useful in a Y86 processor?**
  - Yes, because we have both alternatives by the end of Fetch
    - Address of sequential successor to branch
    - Address of branch target
- **In most processors, branch target requires computation.**
  - Target address often encoded as offset from PC, so addition required
  - Extra cycles for address computation reduce benefit of predictor

An Alternative Approach

- **Branch Target Buffer (BTB)**
  - Accessed during Fetch: before op code is known
  - Entries contain more than just branch history



  - If PC matches address of BTB entry, instr. fetched is branch, predictor is used
  - BTA or next PC will be used for fetch address on next cycle
  - When branch executes, branch history and predictor will be updated

Discussion

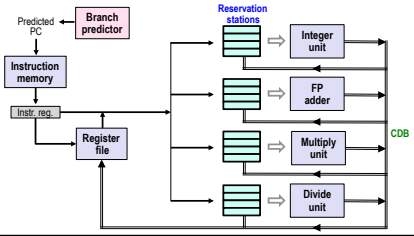
- **Why critical in BTB to match on address?**
  - Why not critical in previously discussed predictors?
- **Some CPUs have included 3 separate predictors**
  1. BTB, accessed in parallel with fetch
  2. History table, accessed later if instr. is branch with no BTB entry
  3. Stack-based predictor for returns (each call pushes a return address)
- **What happens to branch table contents on context switch?**
  - Would it make sense to save and restore with registers?
  - What happens if *your* program runs with table entries from *my* program?
  - OS better not switch processes too often!

Dynamic Scheduling

- **Fundamental idea:**
  - Remove constraint that instructions *execute* in program order.
  - Allow them to execute as soon as operands are available.
  - AKA: *Out-of-order execution*
- **What changes to HW are required to allow this?**
  - Buffers are needed to store instructions waiting for operands
  - Old Decode stage split into two new stages: Issue and Wait for Data
    - Decode: wait to execute until hazards clear and operands available
    - **Issue**: is a buffer available to hold this instruction?
    - **Wait for Data**: wait until operands available
  - Need convenient way to distribute result values to all waiting instructions

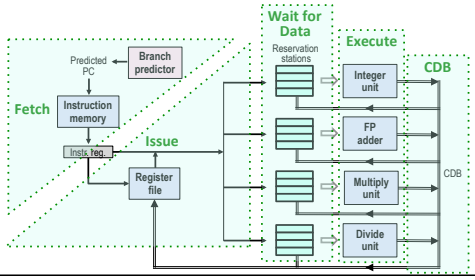
A Tomasulo-based Design

- **We assume F-units discussed previously; new additions to design:**
  - *Reservation stations* (RS) are instruction buffers associated with each F-unit
  - *Common data bus* (CDB) is used to distribute results



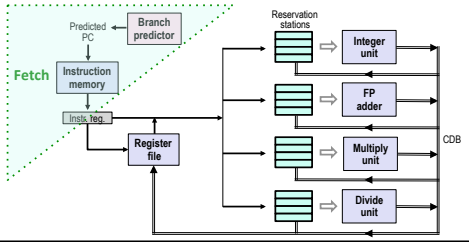
Stage-by-Stage Execution

- **Instructions pass through these stages:**
  - Fetch, Issue, Wait for Data, Execute, CDB



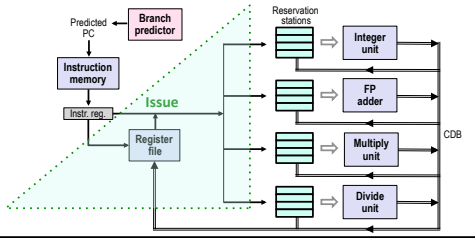
Operational Details

- **Fetch:**
  - The same as PIPE
  - Read instr. bytes, determine next PC



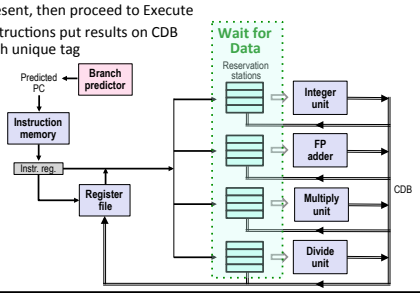
Operational Details

- **Issue:**
  - Read operands from register file, if present; else read *tags* (unique IDs)
  - Determine F-unit, place instr. in empty reservation station (RS)
  - Stall only if no RS available – presence of hazards not a consideration



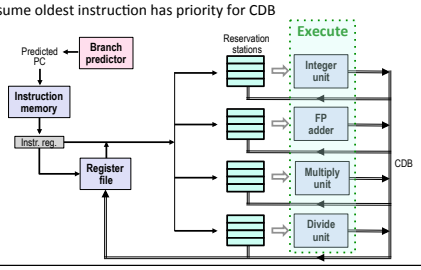
Operational Details

- **Wait for Data:**
  - Wait until all operands are present, then proceed to Execute
  - Instructions put results on CDB with unique tag
  - If RS matches tag, it grabs value



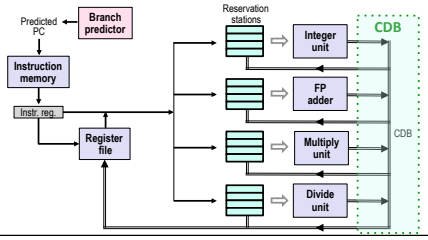
Operational Details

- **Execute:**
  - Normal F-unit operation, 1 or more cycles
  - Instruction must obtain CDB to leave F-unit; otherwise it waits
  - Assume oldest instruction has priority for CDB



Operational Details

- **CDB:**
  - Ordered pair (value, tag) placed on CDB with result of *every* instruction
  - All RS and RF entries that match on tag will read new value



Identifying Result Values

- **Instructions specify operands using register ID**
  - Can hardware use register ID to uniquely identify each result value on CDB?
- **What is max number of *in-flight* instructions at any time?**
  - 5 in integer F-unit (4 in RS, 1 in Execute)
  - 7 in FP adder (4 in RS, 3 in Execute)
  - 9 in multiplier (4 in RS, 5 in Execute)
  - 15 in divider (4 in RS, 11 in Execute, if fully pipelined)
  - Could we ever achieve **36** in-flight instructions simultaneously?
- **How many references to, say, `%rax` in all in-flight instructions?**
  - How can we distinguish between distinct uses of the same register?

Tags: Register Renaming

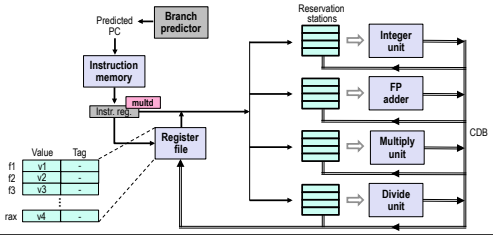
- **Register ID is not enough**
  - Can't distinguish between distinct values of register over time
- **Solution: register renaming**
  - Generate a unique **tag** value for the result of every instruction
    - Could generate using simple counter in Issue – like the DMV
  - After issue, no register IDs/names used – only tags
- **Operation**
  - Every register in register file has value + tag fields
  - If value not present, tag gives ID of instruction producing the most recent value
  - On Issue, grab value if present, else grab tag
  - Wait in reservation station until desired tag shows up on CDB: read associated result value

Example (Cycle 1)

```
multd %f1,%f2
addd %f2,%f3
cvtq2d %rax,%f2
```

F

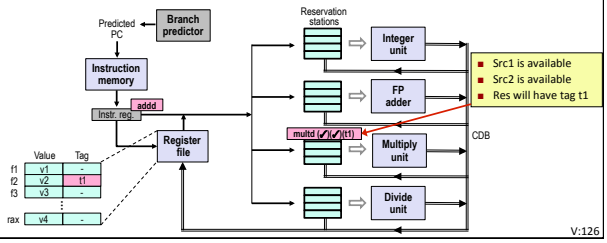
Assumptions:
- multd: 5 cycle latency
- addd: 3 cycle latency
- cvtq2d: 2 cycle latency (uses FP adder)



Example (Cycle 2)

```
multd %f1,%f2
addd %f2,%f3
cvtq2d %rax,%f2
```

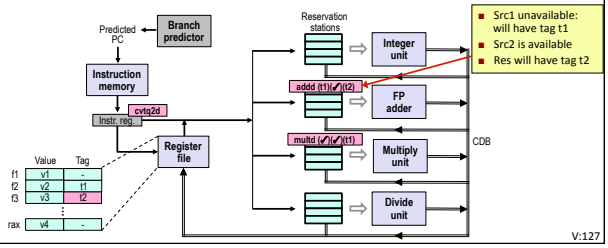
F | IS
F



Example (Cycle 3)

```
multd %f1,%f2
addd %f2,%f3
cvtq2d %rax,%f2
```

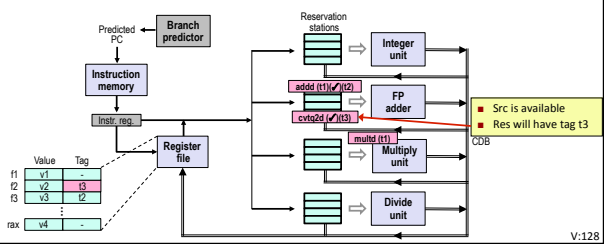
	F	IS	WD
		F	IS
			F



Example (Cycle 4)

```
multd %f1,%f2
addd %f2,%f3
cvtq2d %rax,%f2
```

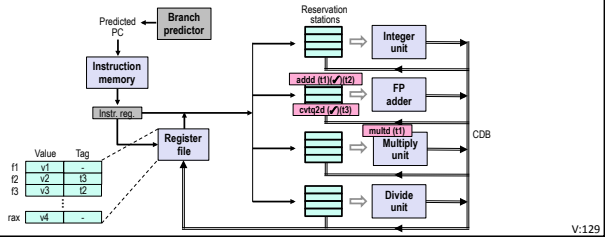
	F	IS	WD	E
		F	IS	WD
			F	IS



Example (Cycle 5)

```
multd %f1,%f2
addd %f2,%f3
cvtq2d %rax,%f2
```

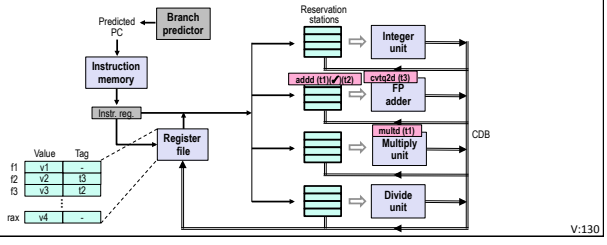
	F	IS	WD	E	E
		F	IS	WD	WD
			F	IS	WD



Example (Cycle 6)

```
multd %f1,%f2
addd %f2,%f3
cvtq2d %rax,%f2
```

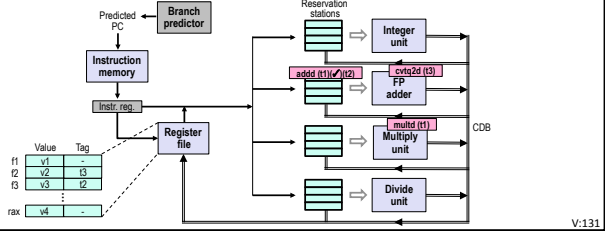
	F	IS	WD	E	E	E
		F	IS	WD	WD	WD
			F	IS	WD	E



Example (Cycle 7)

```
multd %f1,%f2
addd %f2,%f3
cvtq2d %rax,%f2
```

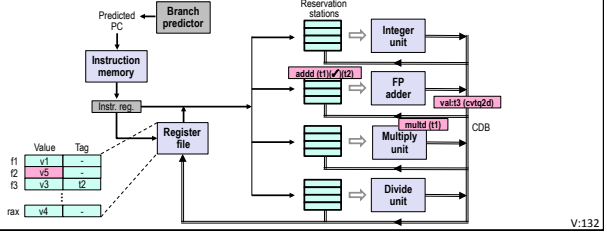
	F	IS	WD	E	E	E	E
		F	IS	WD	WD	WD	WD
			F	IS	WD	E	E



Example (Cycle 8)

```
multd %f1,%f2
addd %f2,%f3
cvtq2d %rax,%f2
```

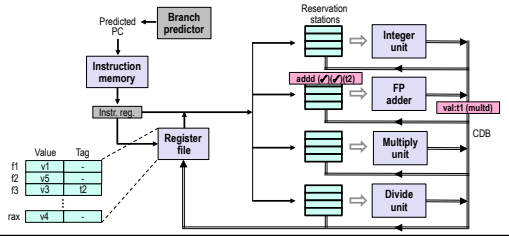
	F	IS	WD	E	E	E	E	E
		F	IS	WD	WD	WD	WD	WD
			F	IS	WD	E	E	CDB



Example (Cycle 9)

```
multd %f1,%f2
addd %f2,%f3
cvtq2d %rax,%f2
```

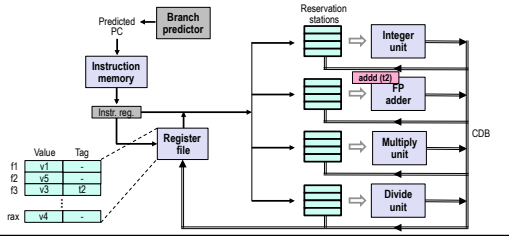
F	IS	WD	E	E	E	E	E	CDB
	F	IS	WD	WD	WD	WD	WD	WD
		F	IS	WD	E	E	CDB	



Example (Cycle 10)

```
multd %f1,%f2
addd %f2,%f3
cvtq2d %rax,%f2
```

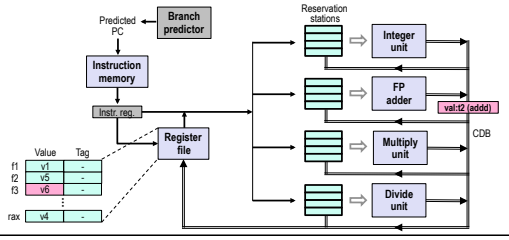
F	IS	WD	E	E	E	E	E	CDB	
	F	IS	WD	WD	WD	WD	WD	WD	E
		F	IS	WD	E	E	CDB		



Example (Cycles 11-13)

```
multd %f1,%f2
addd %f2,%f3
cvtq2d %rax,%f2
```

F	IS	WD	E	E	E	E	E	CDB				
	F	IS	WD	WD	WD	WD	WD	WD	E	E	E	CDB
		F	IS	WD	E	E	CDB					



Discussion

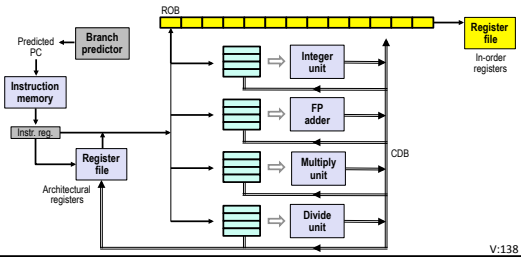
- **For example on previous slides**
  - How was RAW hazard handled?
  - How was WAW hazard handled?
  - Did register renaming simplify the handling of these hazards?
- **Good place to start in understanding any new processor:**
  - How does it handle RAW and WAW dependencies?

So Far So Good

- **Status of current dynamic scheduling processor**
  - Efficiently executes instruction sequence, avoiding stalls
  - Individual instructions wait until their operands are available
  - Register renaming makes WAW hazards vanish
- **But executing instructions out of program order makes it:**
  - Very difficult to recover from mispredicted branches
  - Very difficult to support precise exceptions
- **Both of these depend on original program order of instructions**
  - How can we execute instructions out-of-order and still maintain a record of program order?

Solution

- **Add a Re-order Buffer (ROB)**
  - Instructions placed in ROB in order at Issue, "retire" from ROB in order
  - Strict FIFO, buffers results from execution, updates "in-order" register file



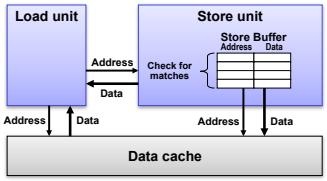
Operation with ROB

- **Mispredicted branches**
  - When mispredicted branch retires, throw away all following ROB entries
  - Restart fetch at correct address
  - Restore "architectural" registers to contents of "in-order" registers
    - Exactly what they would be at this point in sequential execution
- **Precise exceptions**
  - If instruction with exception retires, throw away all other ROB entries
  - "In-order" register file contains context at point of exception
- **In effect, we've added an in-order layer around the out-of-order core**
  - Gives us best of both worlds, but with increased misprediction penalty

Memory Considerations

- **Can hazards occur on memory accesses? Let's explore:**
  - **Loads** should execute as quickly as possible
    - Later instructions will be waiting for the result
    - Can go as soon as address register is available
    - No harm if this instruction needs to be "undone" later
  - In contrast, **stores** modify memory: much harder to "undo"
    - Must ensure no earlier mispredicted branch or instr. with exception
    - Wait until store retires to change memory (hence in program order)
  - What ensures that load won't get obsolete value from memory?
    - Could be earlier store waiting to change that location
- **Solution: special load-store unit to handle this**
  - Really another F-unit operating in parallel with others

Load and Store Units



- **Store buffer contains (address, data) pairs**
  - Entry created when write issued, retained until write retires
  - At retirement, write modifies memory; entry in store buffer removed
- **Load unit must check store buffer for matching address on every access**
  - If address matches, grab corresponding data (or wait until value available)
  - Less obvious: if *address* not yet computed for *any* store, load must wait

Lab 5: Dynamic Scheduling

- **You are given**
  - Assembler + cycle-accurate simulator for Y86-64 CPU using dynamic scheduling
    - Y86-64 augmented with two integer instructions: multq and addq
- **You write pathological 20 instruction program (no jumps, calls, rets)**
  - Takes as long to run as possible (use long latency instructions, long dependence chains)
  - At least one instance of each of these:
    - Pipeline stall
    - RAW hazard
    - WAW hazard
    - Functional unit conflict
    - CDB conflict
    - Load forced to wait
    - Load gets value from store buffer
    - Loads execute out-of-order

Lab 5

- **Motivation**
  - Understand how dynamically scheduling really works (without having to design a CPU that uses it)
  - Claim: you can't create code with the required pathologies without a reasonable understanding of how the CPU works
- **Discussion**
  - Must avoid any exceptions (bad memory addresses, divide by zero)
  - Can use the simulator to observe the execution of your code
  - Lab score based on
    - Points for generating at least one instance of each inefficiency
    - Points for overall cycle time – must be above threshold for full credit

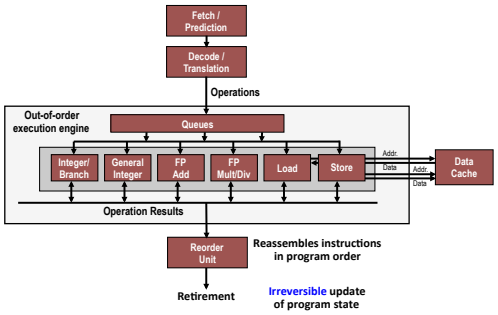
One Last Consideration

- **We'd be done if we were using a RISC instruction set**
  - Outcome of conditionals sets general purpose register to 0 or 1
  - Conditional branches and moves get condition from GP register
- **As HW reorders instructions, dependencies on condition codes must be considered**
  - Bottom line: values of CC must be tagged, passed around, and updated just like other registers

Intel ISA Complexity

- **How does an Intel CPU do all this fast enough to give good performance?**
  - Their out-of-order execution core is wrapped within both
    - An in-order layer that uses a ROB
    - Another layer that **translates** complex x86 instructions into simpler operations that resemble RISC instructions
  - In effect, their out-of-order execution core is executing RISC instructions!

Intel CPUs: Perspective #1



Intel CPUs: Perspective #2

- **Pentium 4**

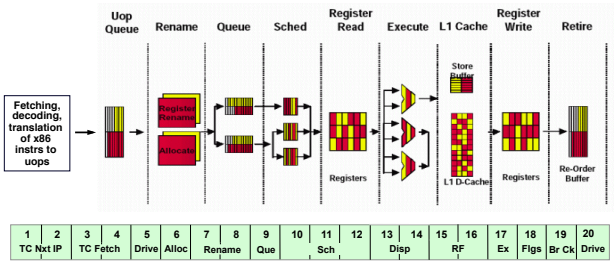


Image from www.xbitlabs.com