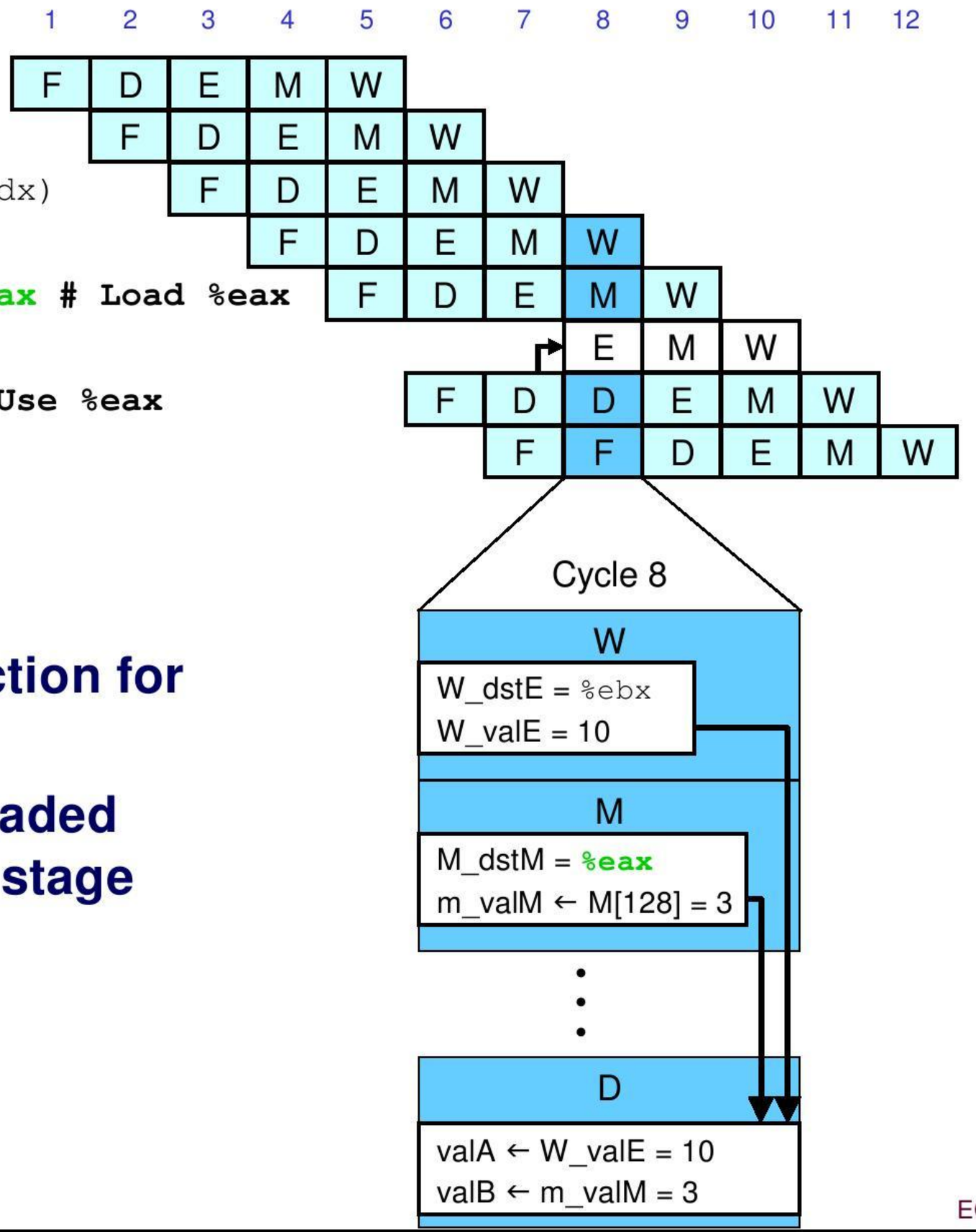
Avoiding Load/Use Hazard

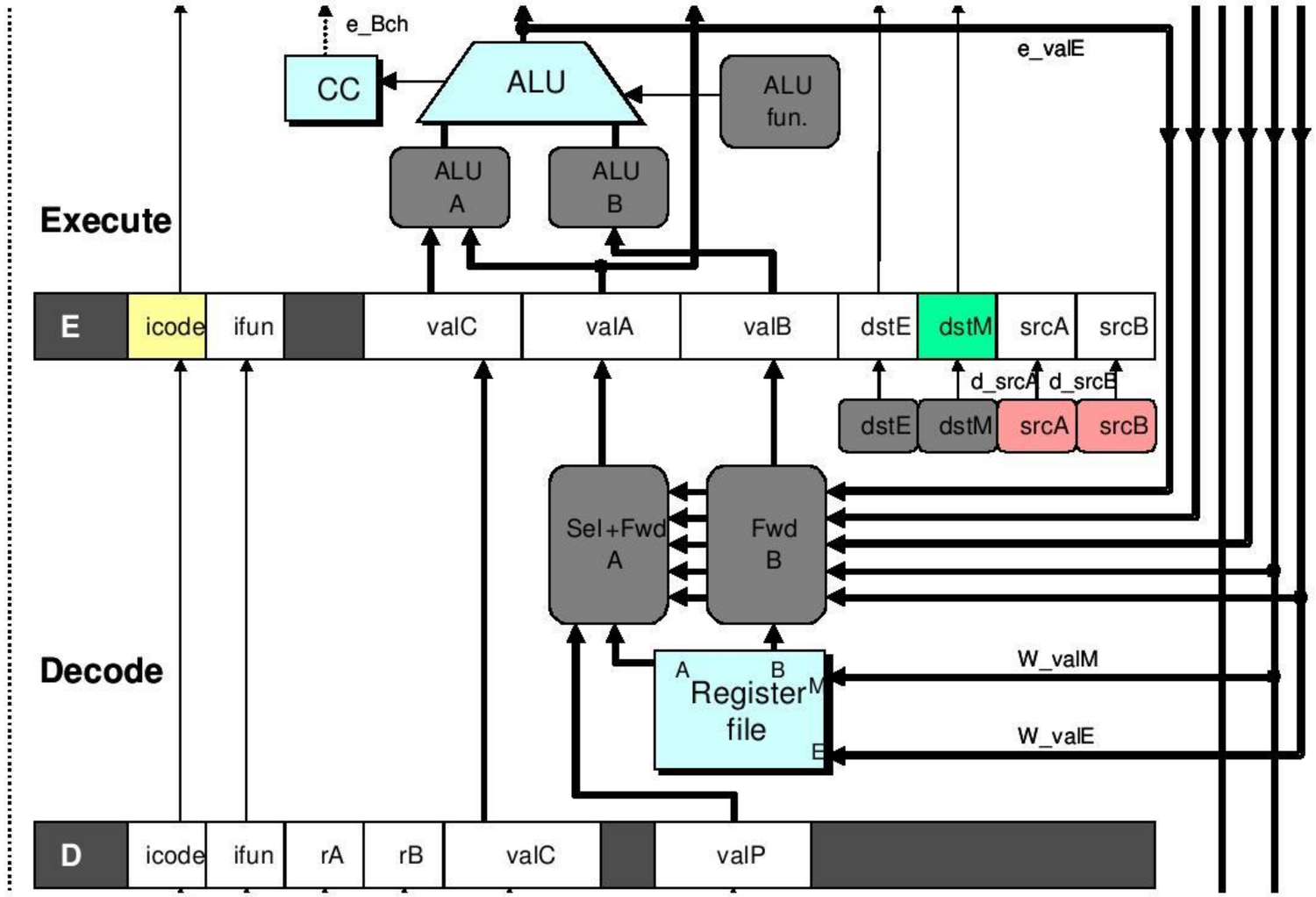
```
# demo-luh.ys
0x000: irmovl $128,%edx
0x006: irmovl  $3,%ecx
0x00c: rmmovl %ecx, 0(%edx)
0x012: irmovl $10,%ebx
0x018: mrmovl 0(%edx),%eax # Load %eax
       bubble
0x01e: addl %ebx,%eax # Use %eax
0x020: halt
```



Can't beat this:

- Stall reading instruction for one cycle
- Can then forward loaded value from memory stage

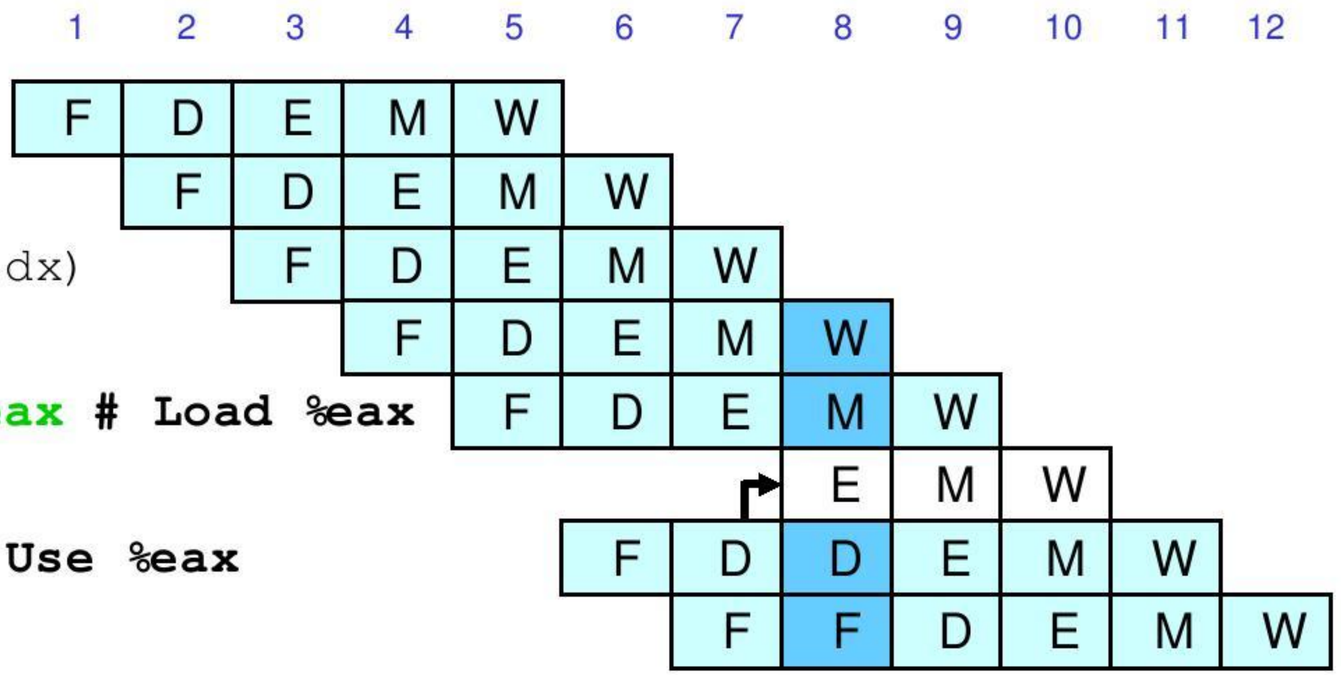
Detecting Load/Use Hazard



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

Control for Load/Use Hazard

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



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

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

Branch Misprediction Example

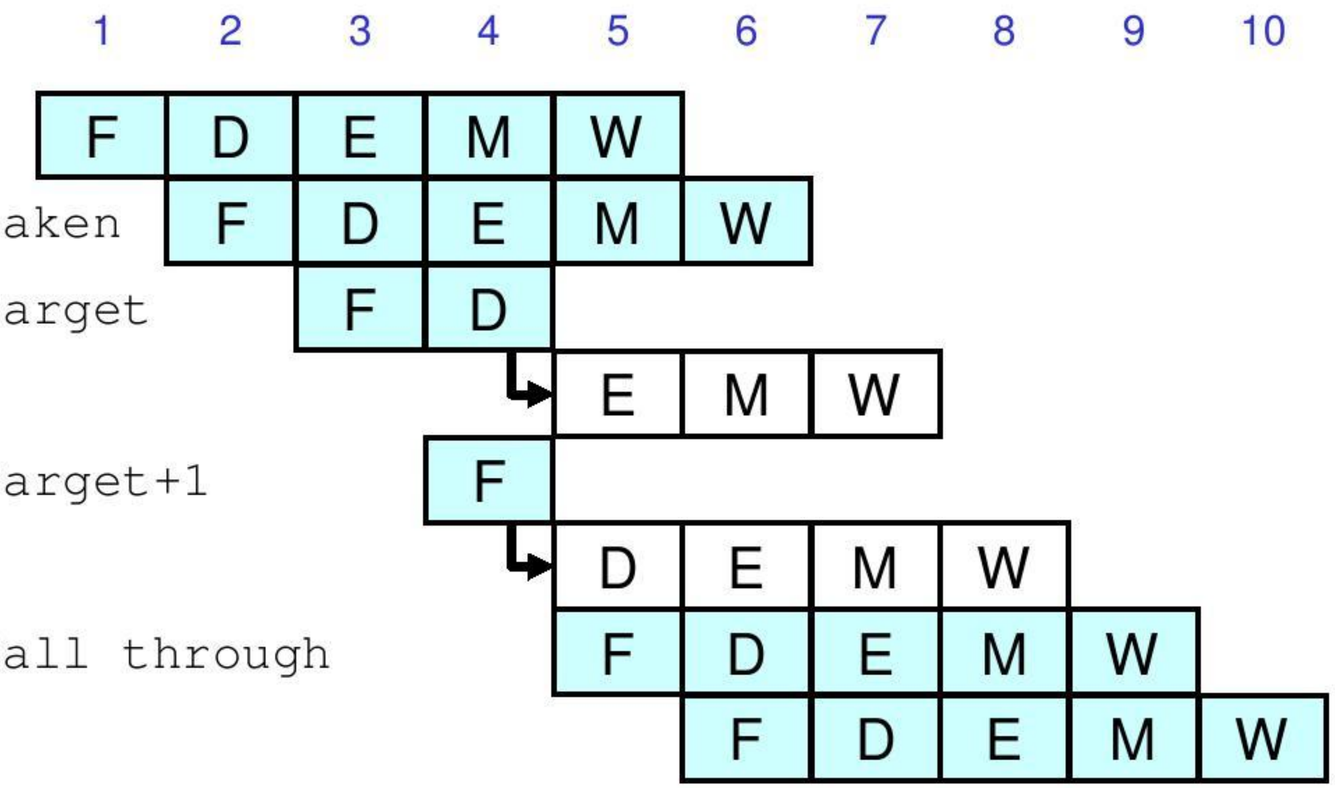
demo-j.ys

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

- Should only execute first 7 instructions
- Our branch predictor will mispredict the branch as taken

Handling Misprediction

```
# demo-j.ys
0x000:      xorl %eax,%eax
0x002:      jne target # Not taken
0x011: t:   irmovl $2,%edx # Target
            bubble
0x017:      irmovl $3,%ebx # Target+1
            bubble
0x007:      irmovl $1,%eax # Fall through
0x00d:      nop
```



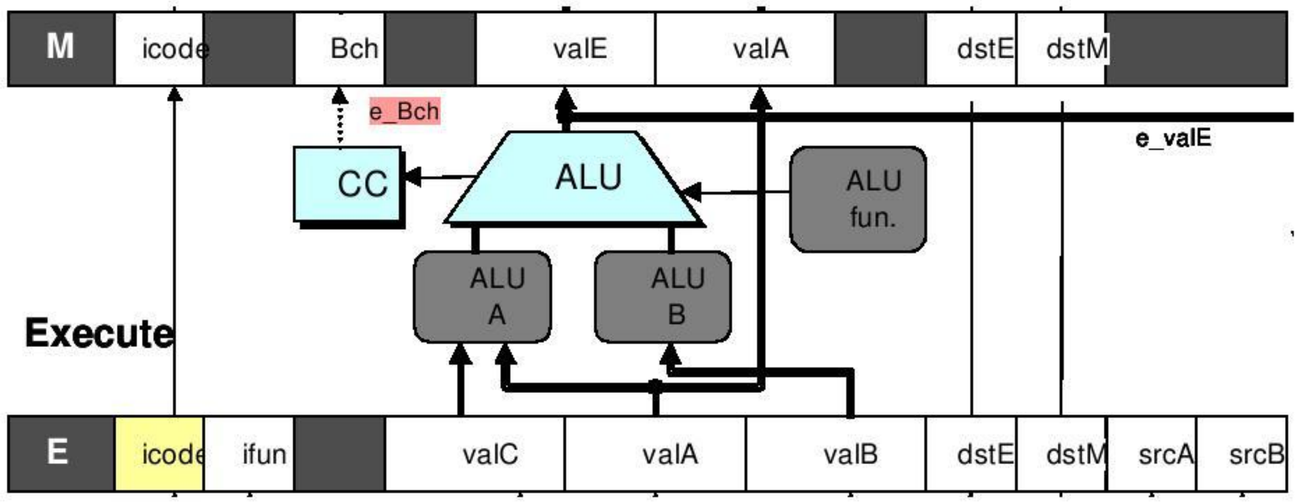
Branch misprediction detected in execute stage

- Instruction at target address is in decode
- Next instruction has been fetched

Recovering

- Detect branch not-taken in execute stage
- On next cycle, replace mispredicted instructions by bubbles
- We're lucky! No side effects have occurred yet

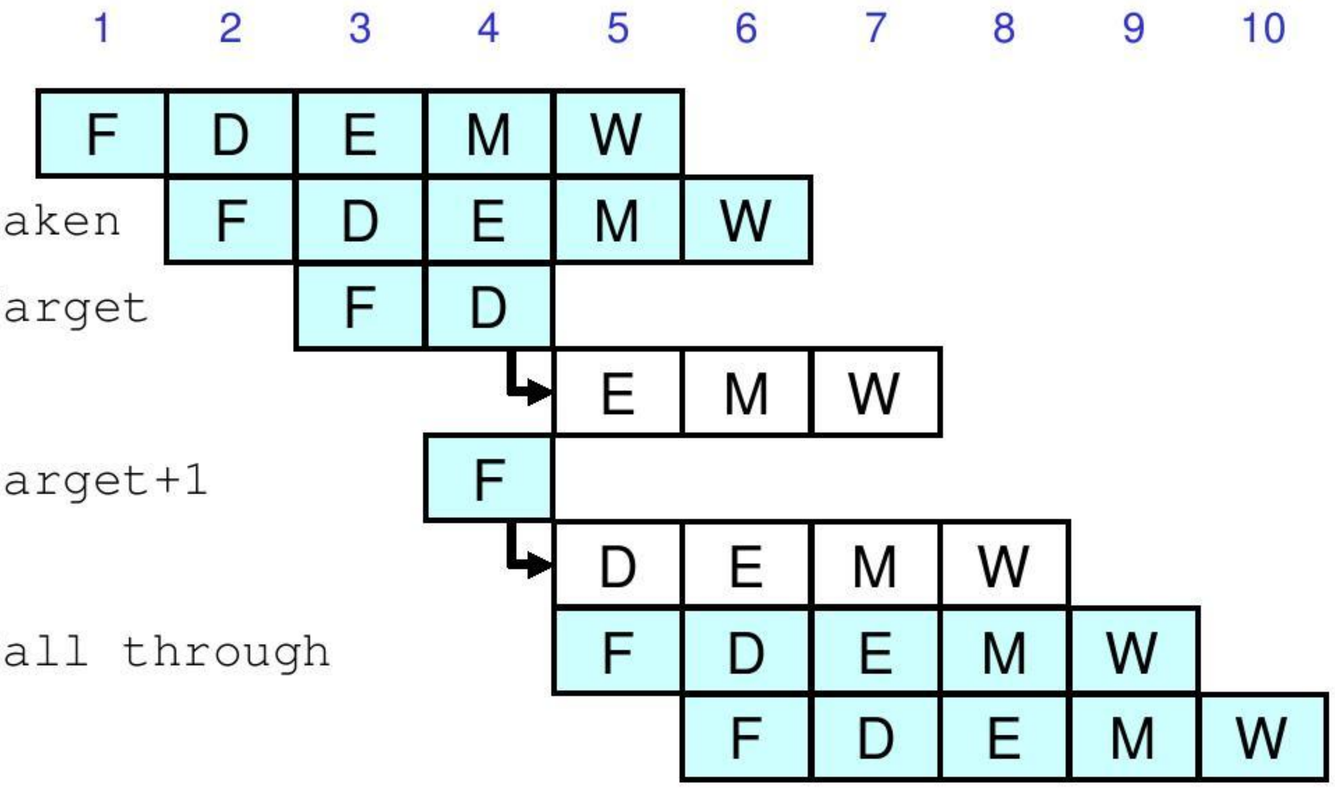
Detecting Mispredicted Branch



Condition	Trigger
Mispredicted Branch	E_icode = IJXX & !e_Bch

Control for Misprediction

```
# demo-j.ys
0x000:      xorl %eax,%eax
0x002:      jne target # Not taken
0x011: t:   irmovl $2,%edx # Target
            bubble
0x017:      irmovl $3,%ebx # Target+1
            bubble
0x007:      irmovl $1,%eax # Fall through
0x00d:      nop
```



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

Return Example

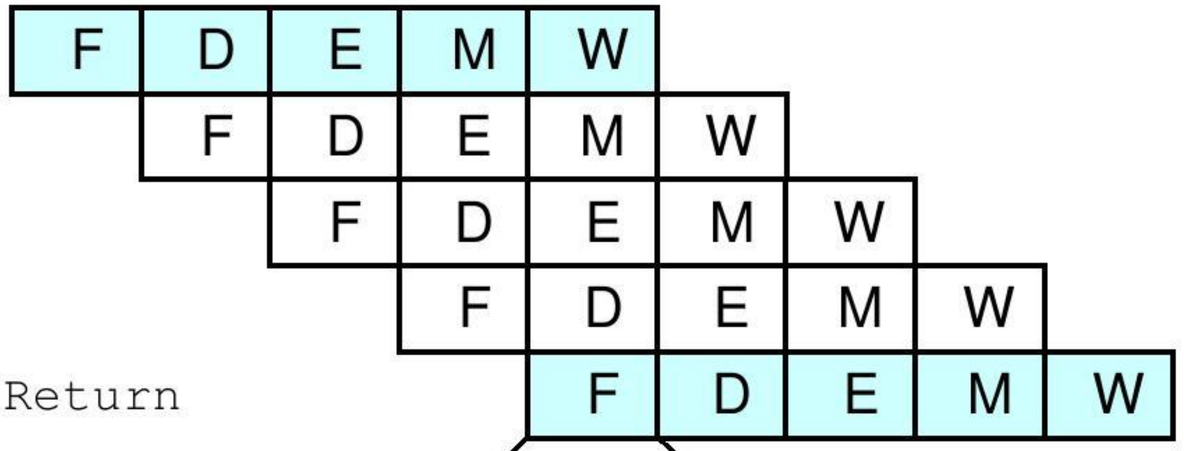
demo-retb.ys

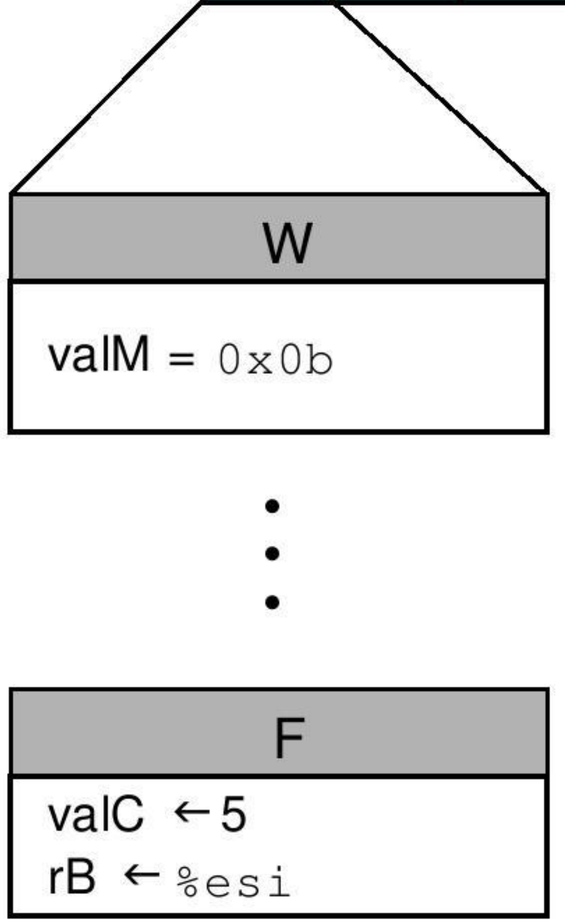
```
0x000:     irmovl Stack,%esp  # Initialize stack pointer
0x006:     call p             # Procedure call
0x00b:     irmovl $5,%esi     # Return point
0x011:     halt
0x020: .pos 0x20
0x020: p:  irmovl $-1,%edi    # procedure
0x026:     ret
0x027:     irmovl $1,%eax     # Should not be executed
0x02d: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

- Without special handling we would execute three more instructions after `ret` before we know return address
- Instead, we'll freeze the pipe when `ret` is fetched

Correct Return Example

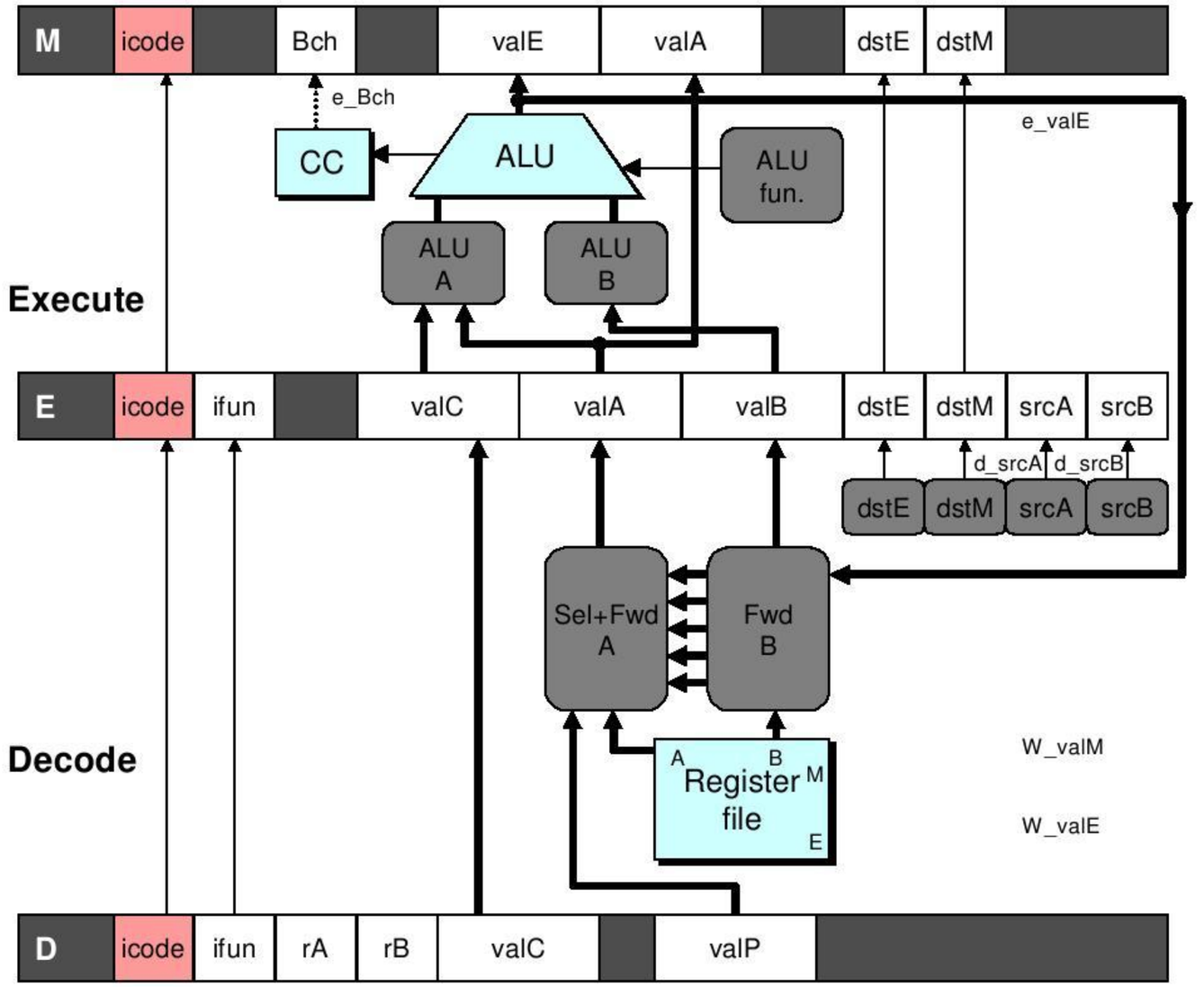
```
# demo-retb
0x026:      ret
            bubble
            bubble
            bubble
0x00b:      irmovl $5,%esi # Return
```





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

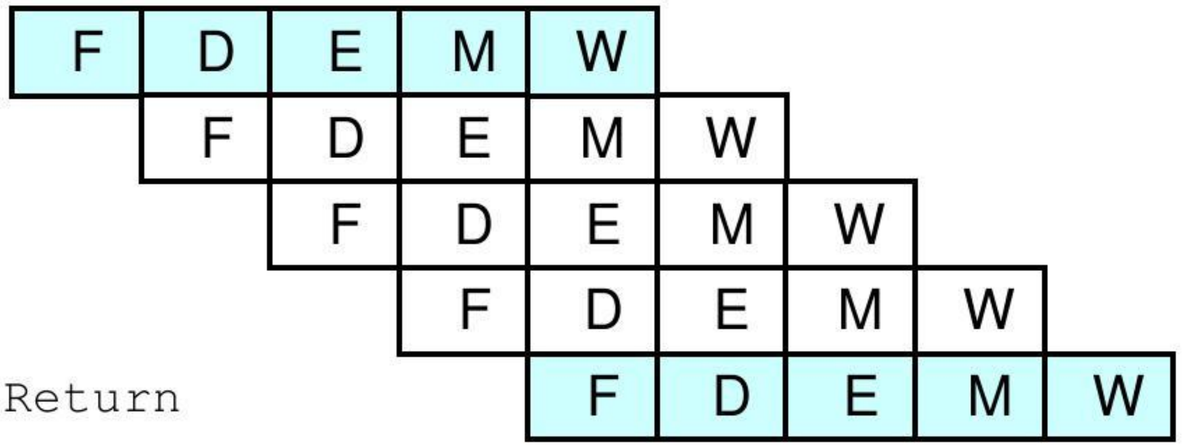
Detecting Return



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

Control for Return

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



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

Special Control Cases

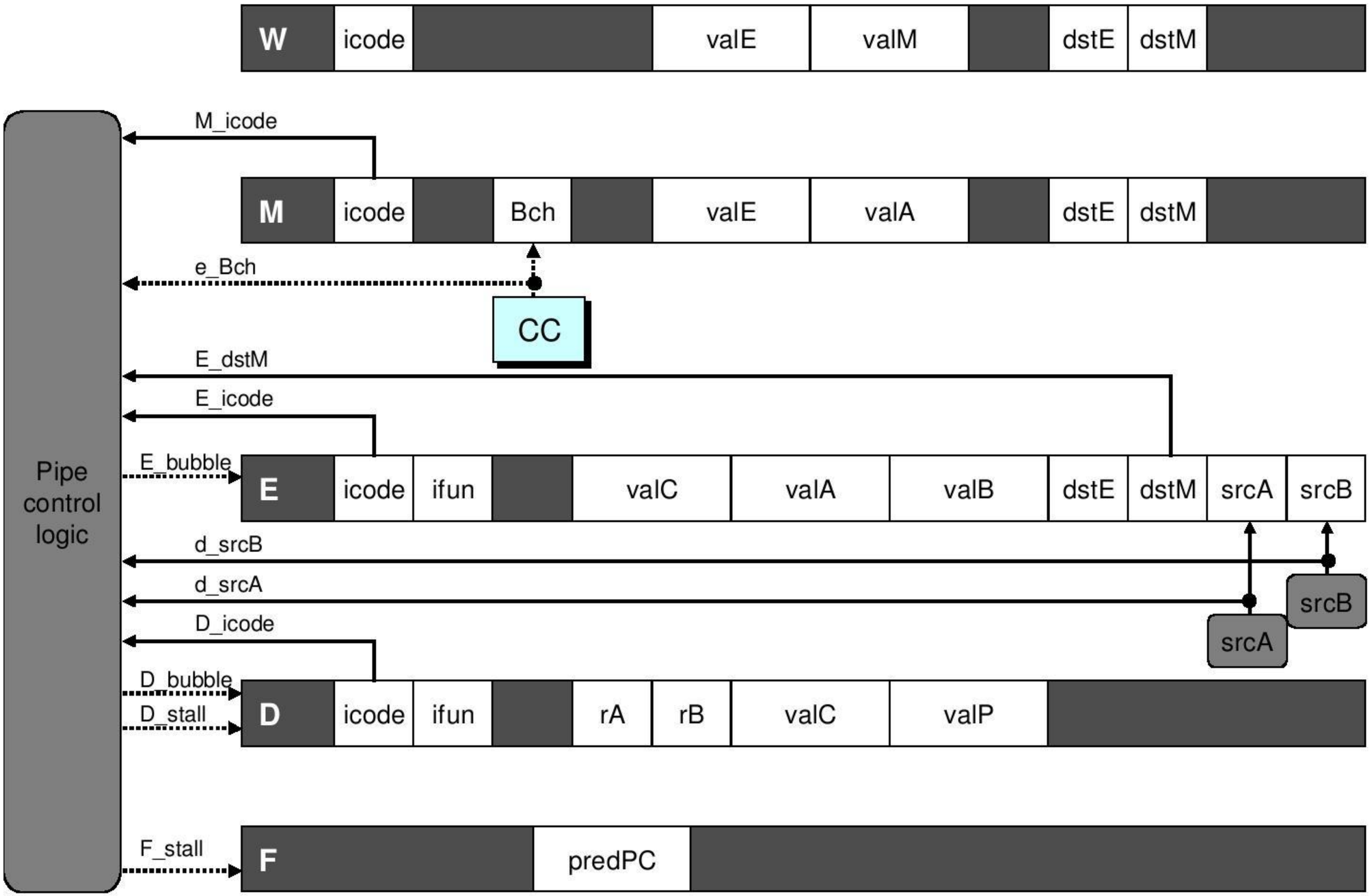
Detection

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

Action (on next cycle)

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

Implementing Pipeline Control



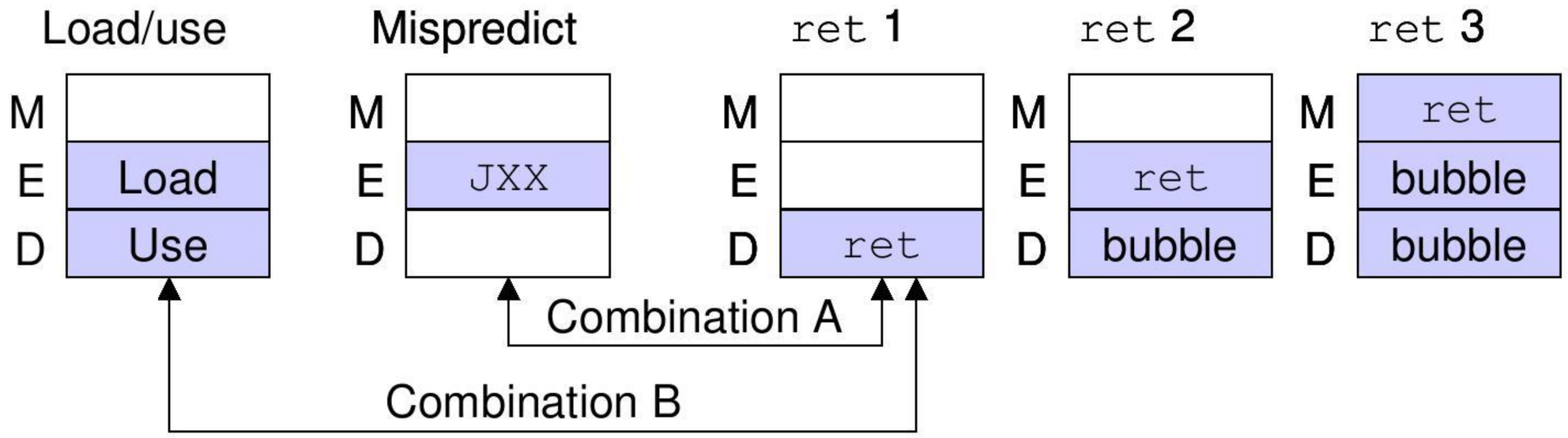
- Combinational logic generates pipeline control signals
- Action occurs at start of following cycle

Initial Version of Pipeline Control

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

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

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

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

How do I know this works?

Control Combinations



- Special cases that can arise on same clock cycle

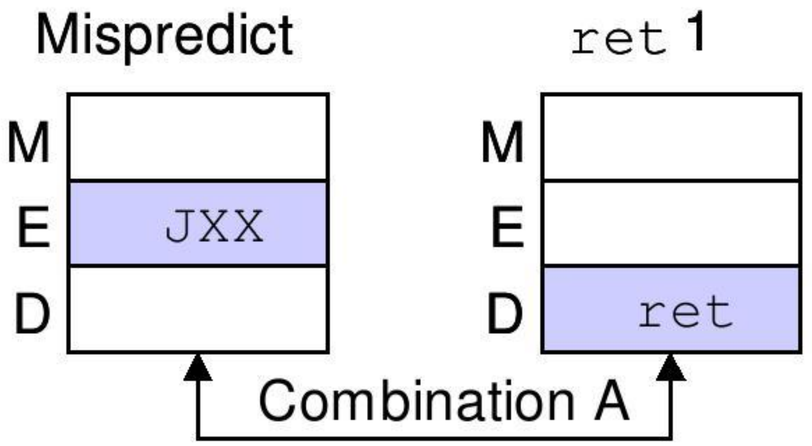
Combination A

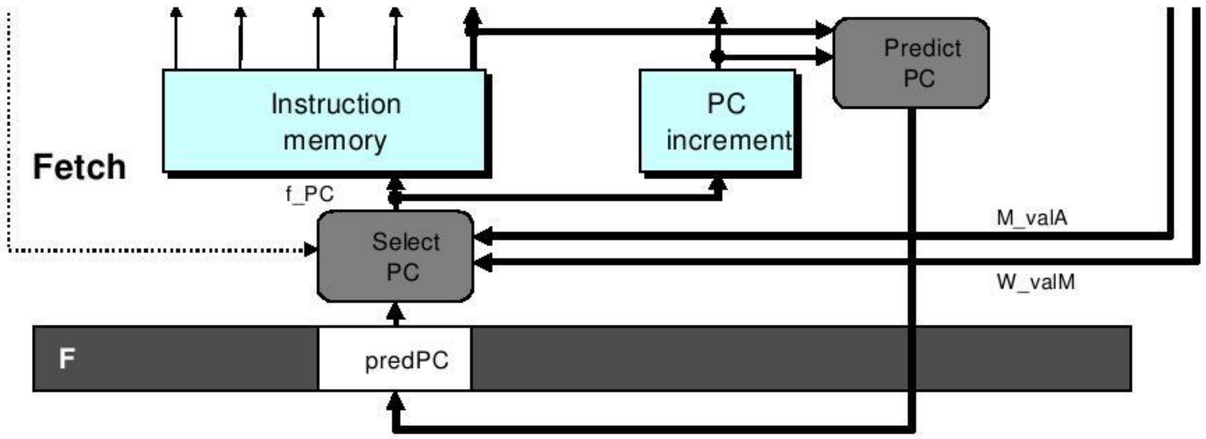
- Not-taken branch
- `ret` instruction at branch target

Combination B

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

Control Combination A





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

- Should handle as mispredicted branch
- Combination will also stall F pipeline register
- But PC selection logic will be using M_valM anyway
- Correct action 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 attempt to bubble *and* stall pipeline register D
  - Would be signaled by processor as pipeline error
- Combination not handled correctly in authors' initial version
  - But it passed many simulation tests; caught only with systematic analysis

Handling Control Combination B

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
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 { IMRMOVL, IPOPL }
            && E_dstM in { d_srcA, d_srcB });
```

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

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

Performance Metrics

Clock rate

- Measured in Megahertz or Gigahertz
- Function of stage partitioning and circuit design
  - Keep amount of work per stage small

Rate at which instructions executed

- CPI: cycles per instruction
- On average, how many clock cycles does each instruction require?
- Function of pipeline design and benchmark programs
  - E.g., how frequently are branches mispredicted?

CPI for PIPE

Ideal CPI = 1.0

- Fetch instruction each clock cycle
- Process new instruction every cycle
  - Although each individual instruction has latency of 5 cycles

Actual CPI > 1.0

- Sometimes pipeline stalls, branches are mispredicted

Computing CPI

- C clock cycles
- I instructions executed to completion
- B bubbles injected (C = I + B)

$$CPI = C/I = (I+B)/I = 1.0 + B/I$$

- Factor B/I represents average penalty (per instruction) due to bubbles

CPI for PIPE (Cont.)

$$B/I = LP + MP + RP$$

Typical Values

- **LP: Penalty due to load/use hazard stalling**
  - 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!)

State-of-the-Art Pipelining

What have we ignored in our Y86 implementation?

- Balancing delay in each stage
  - Which stage is longest, how might we speed it up?
- Handling exceptions
  - When something comes up hardware can't handle, it hands off control to operating system
  - Mechanism essentially the same as that used for interrupts
  - Design to support what software expects (*precise exceptions*) has implications in design of pipeline

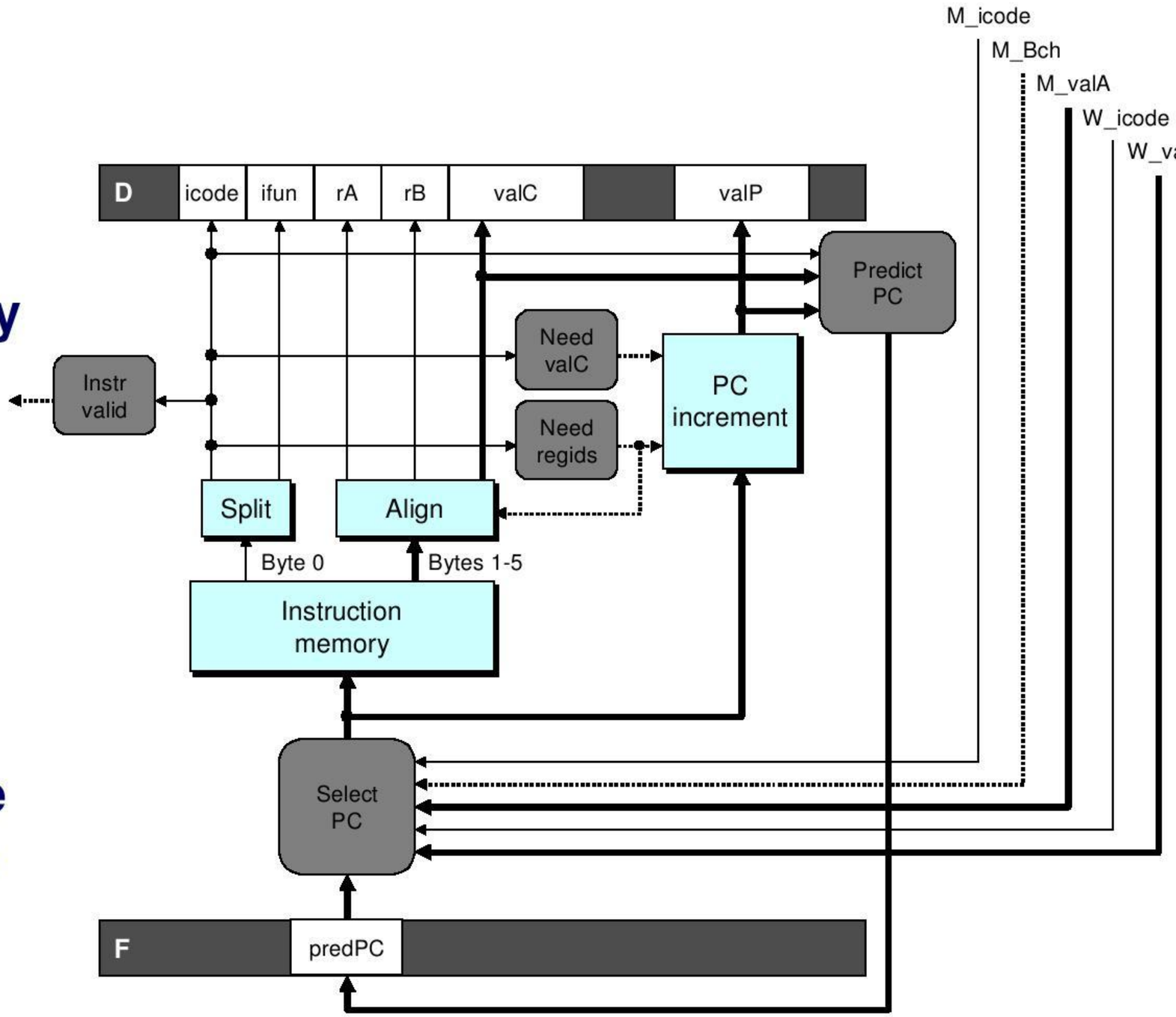
Fetch Logic Revisited

During Fetch Cycle

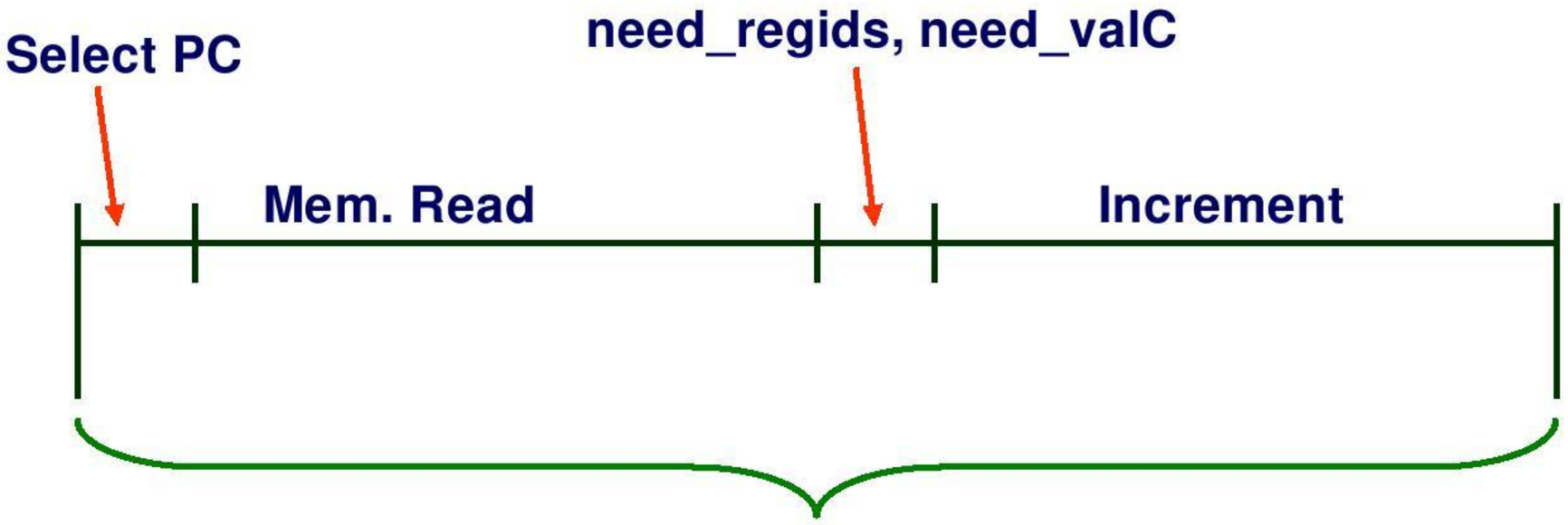
1. Select PC
2. Read bytes from instruction memory
3. Examine icode to determine instruction length
4. Increment PC

Timing

- Steps 2 & 4 require significant amount of time

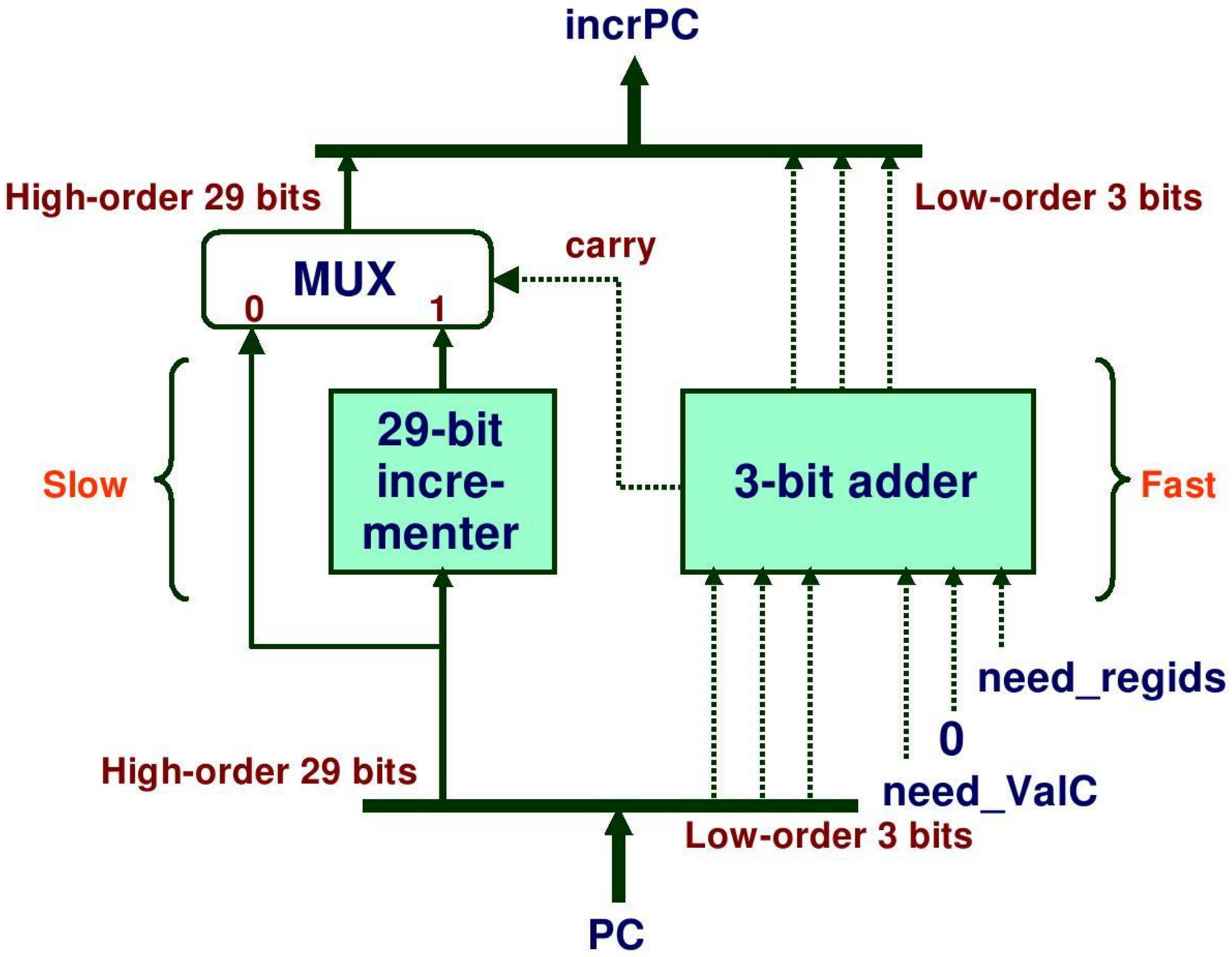


Standard Fetch Timing

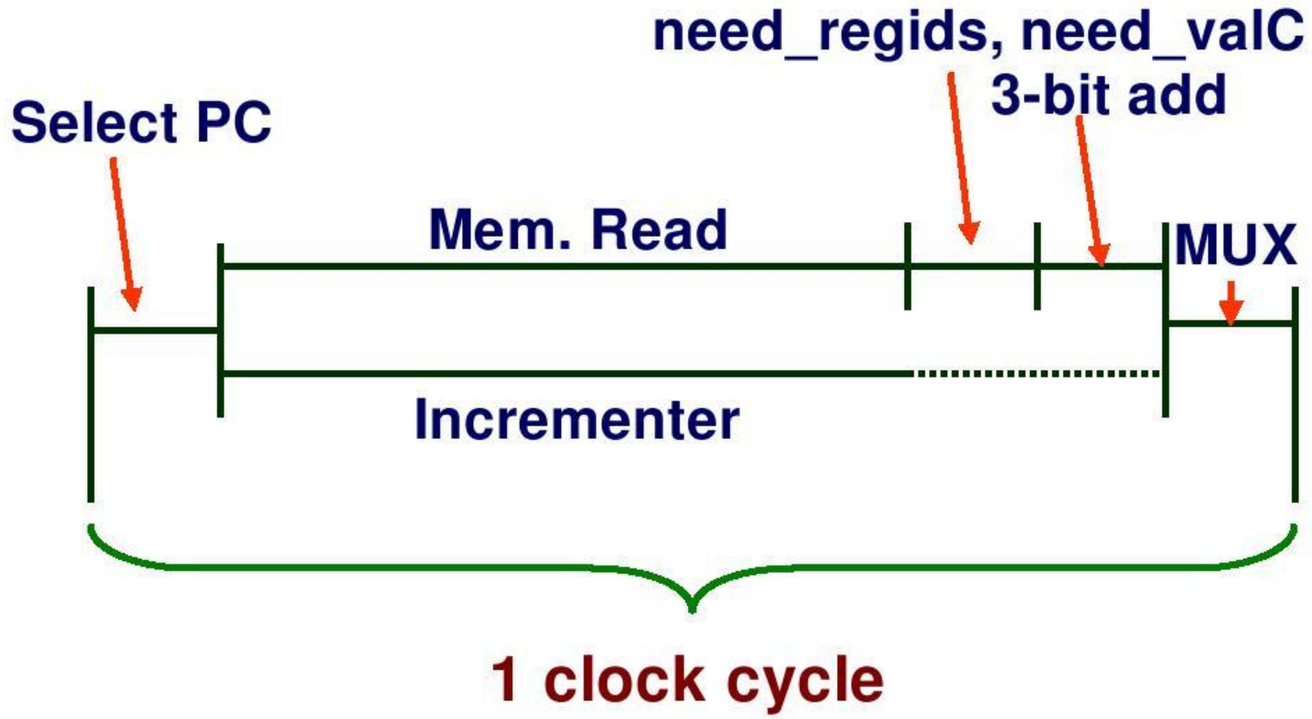


- Must Perform Everything in Sequence
- Can't compute incremented PC until know how much to increment it by

A Fast PC Increment Circuit



Modified Fetch Timing



Standard cycle

29-Bit Incrementer

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

Exceptions

- Conditions under which pipeline cannot continue normal operation

Causes

- Halt instruction (Current)
- Bad address for instruction or data (Previous)
- Invalid instruction (Previous)
- Pipeline control error (Previous)

Desired Action

- Complete some instructions
  - Either current or previous (depends on exception type)
- Discard others
- Call (transfer control to) exception handler
  - Like an unexpected procedure call

Exception Examples

Detect in Fetch Stage

```
jmp $-1                          # Invalid jump target

.byte 0xFF                       # Invalid instruction code

halt                             # Halt instruction
```

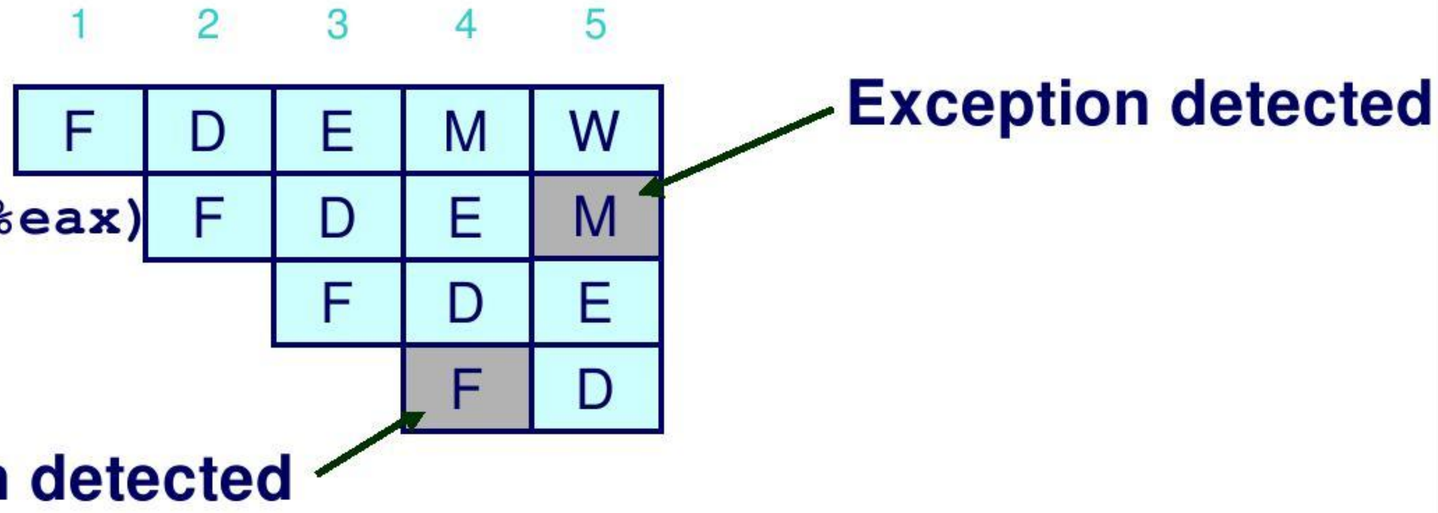
Detect in Memory Stage

```
irmovl $100,%eax
rmmovl %eax,0x10000(%eax) # invalid address
```

Exceptions in Pipeline Processor #1

```
# demo-exc1.ys
irmovl $100,%eax
rmmovl %eax,0x10000(%eax) # Invalid address
nop
.byte 0xFF                # Invalid instruction code
```

```
0x000: irmovl $100,%eax
0x006: rmmovl %eax,0x10000(%eax)
0x00c: nop
0x00d: .byte 0xFF
```



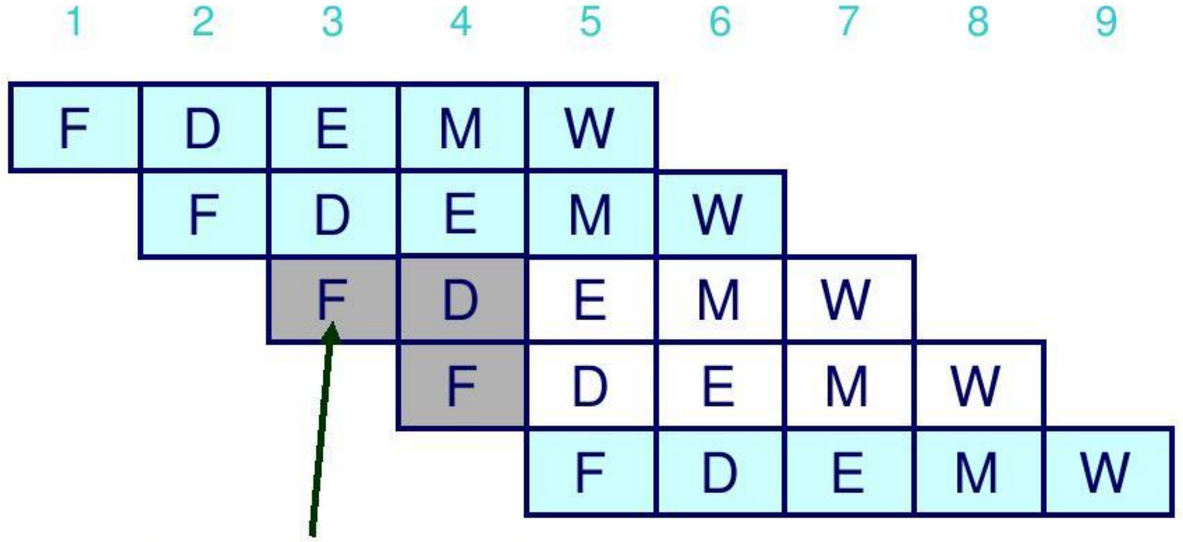
Desired Behavior

- `rmmovl` should cause exception

Exceptions in Pipeline Processor #2

```
# demo-exc2.ys
0x000:     xorl %eax,%eax    # Set condition codes
0x002:     jne t             # Not taken
0x007:     irmovl $1,%eax
0x00d:     irmovl $2,%edx
0x013:     halt
0x014: t:  .byte 0xFF        # Target
```

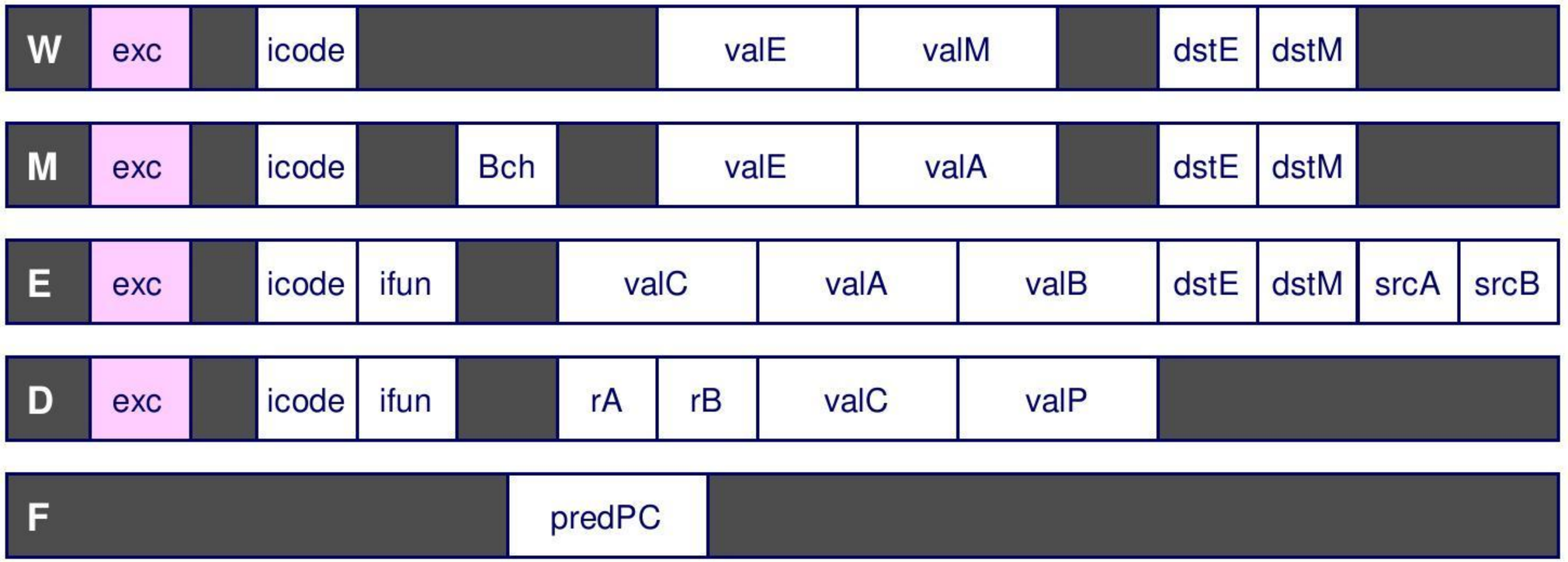
```
0x000:     xorl %eax,%eax
0x002:     jne t
0x014: t:  .byte 0xFF
0x???:  (I'm lost!)
0x007:     irmovl $1,%eax
```



Desired Behavior

- No exception should occur

Maintaining Exception Ordering

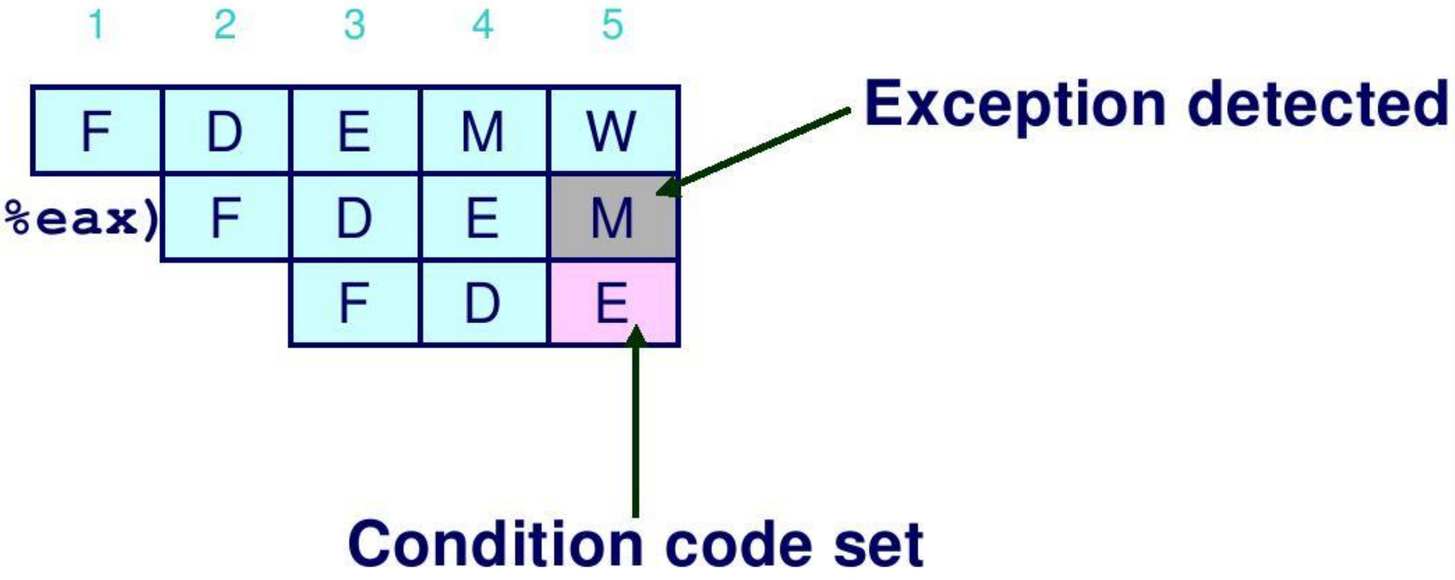


- Add exception status field to pipeline registers
- Fetch stage sets to either "AOK," "ADR" (when bad fetch address), or "INS" (illegal instruction)
- Decode & execute pass values through
- Memory either passes through or sets to "ADR"
- Exception triggered only when instruction hits write back

Side Effects in Pipeline Processor

```
# demo-exc3.ys
irmovl $100,%eax
rmmovl %eax,0x10000(%eax) # invalid address
addl %eax,%eax            # Sets condition codes
```

```
0x000: irmovl $100,%eax
0x006: rmmovl %eax,0x10000(%eax)
0x00c: addl %eax,%eax
```



Desired Behavior

- `rmmovl` should cause exception
- No following instruction should change any state

Avoiding Side Effects

Presence of Exception Should Disable State Update

- When exception detected in memory stage
  - Disable condition code setting in execute
  - Must happen in same clock cycle
- When exception passes to write-back stage
  - Disable memory write in memory stage
  - Disable condition code setting in execute stage

Implementation in Y86 Tools

- Hardwired into the design of the PIPE simulator
- You don't have to worry about it, can't change it

Rest of Exception Handling

Calling Exception Handler

- Push PC onto stack
  - Either PC of faulting instruction or of next instruction
  - Usually pass through pipeline along with exception status
- Jump to handler address
  - Usually fixed address
  - Defined as part of ISA

Implementation

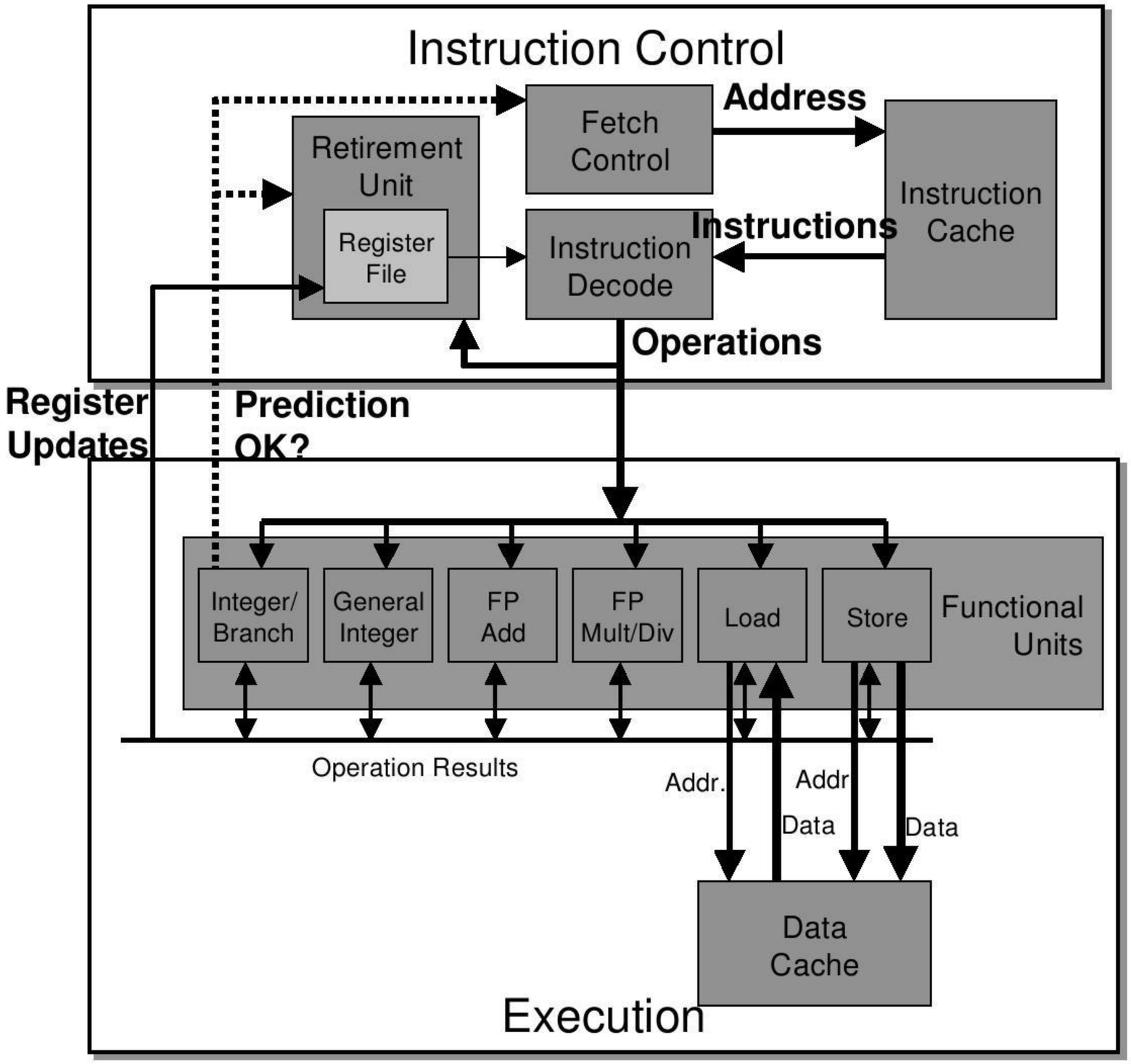
- Critical for real hardware
- Seldom implemented in simulators
  - No OS running to pass control to!

State-of-the-Art Pipelining

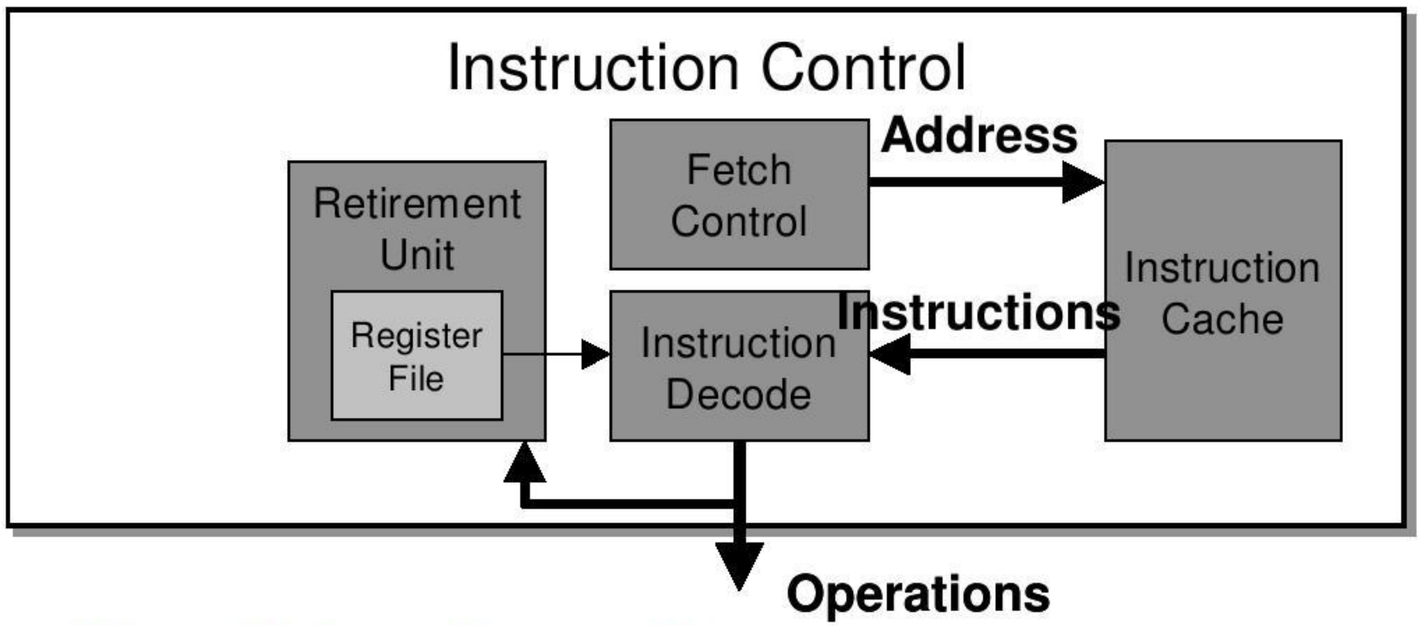
What else have we ignored?

- More complex instructions: consider FP divide, sqrt
  - Take many cycles to execute
  - Forwarding can't solve hazards: more data stalls
  - Important for compiler to schedule code
- Deeper pipelines to allow faster cycle times
  - Increased penalty on misprediction
  - Increased emphasis on branch prediction
- Actual memory hierarchy issues (will increase CPI)
  - Difficult to complete memory access in one cycle!
  - Possibility of cache misses, TLB misses, page faults
- Superscalar approach: process multiple instructions/cycle
- Dynamic scheduling
  - Scheduling = determining instruction execution order
  - Hardware decides based on data dependencies, resources

Modern CPU Design



Instruction Control



Grabs Instruction Bytes From Memory

- Based on Current PC + Predicted Targets for Predicted Branches
- Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

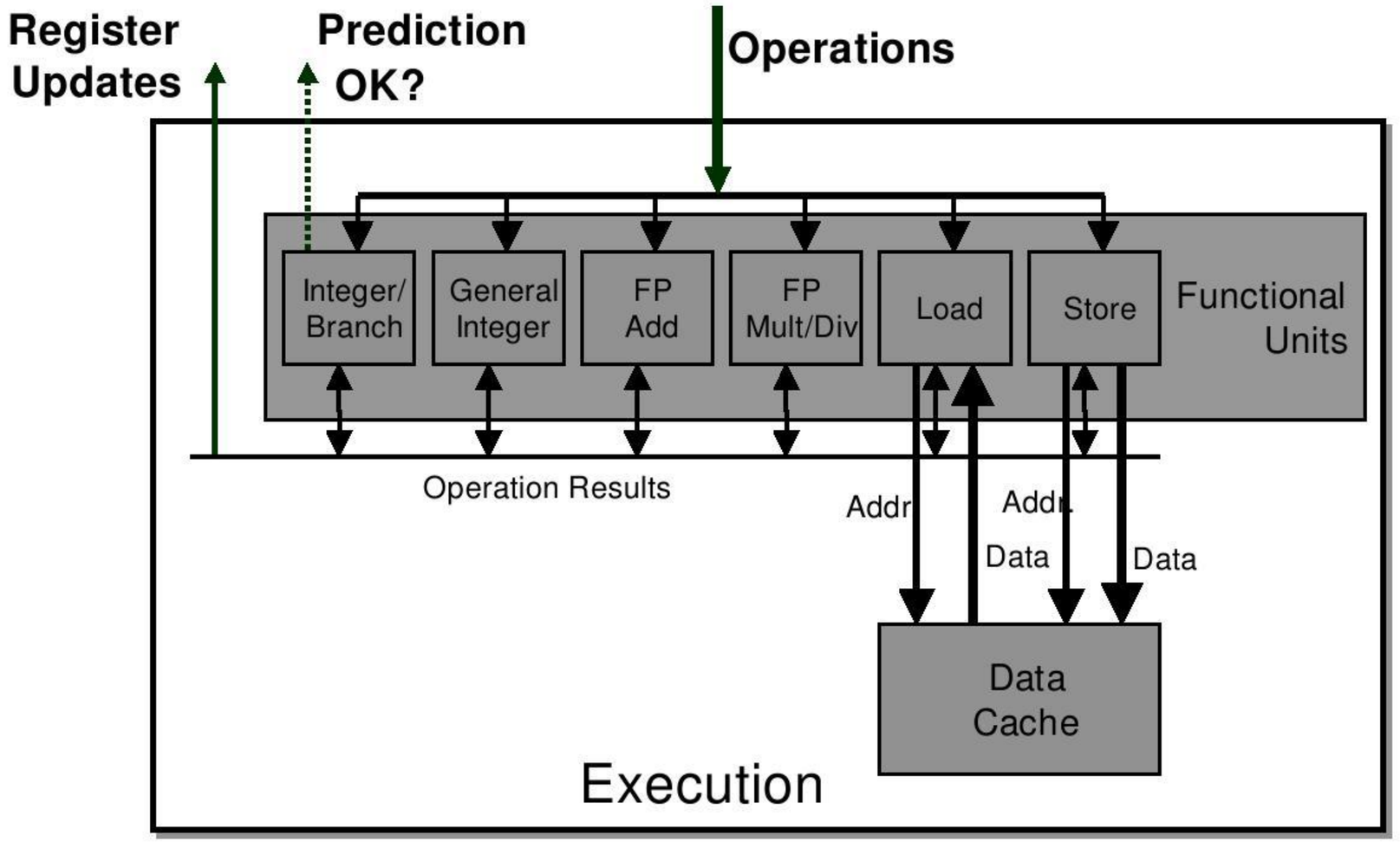
Translates Instructions Into *Operations*

- Primitive steps required to perform instruction
- Typical instruction requires 1–3 operations

Converts Register References Into *Tags*

- Abstract identifier linking destination of one operation with sources of later operations

Execution Unit



- Multiple functional units
  - Each can operate in independently
- Operations execute as soon as operands available
  - Not necessarily in program order
  - Within limits of functional units
- Control logic
  - Ensures behavior equivalent to sequential program execution

CPU Capabilities of Pentium III

Multiple Instructions Can Execute in Parallel

- 1 load
- 1 store
- 2 integer (one may be branch)
- 1 FP Addition
- 1 FP Multiplication or Division

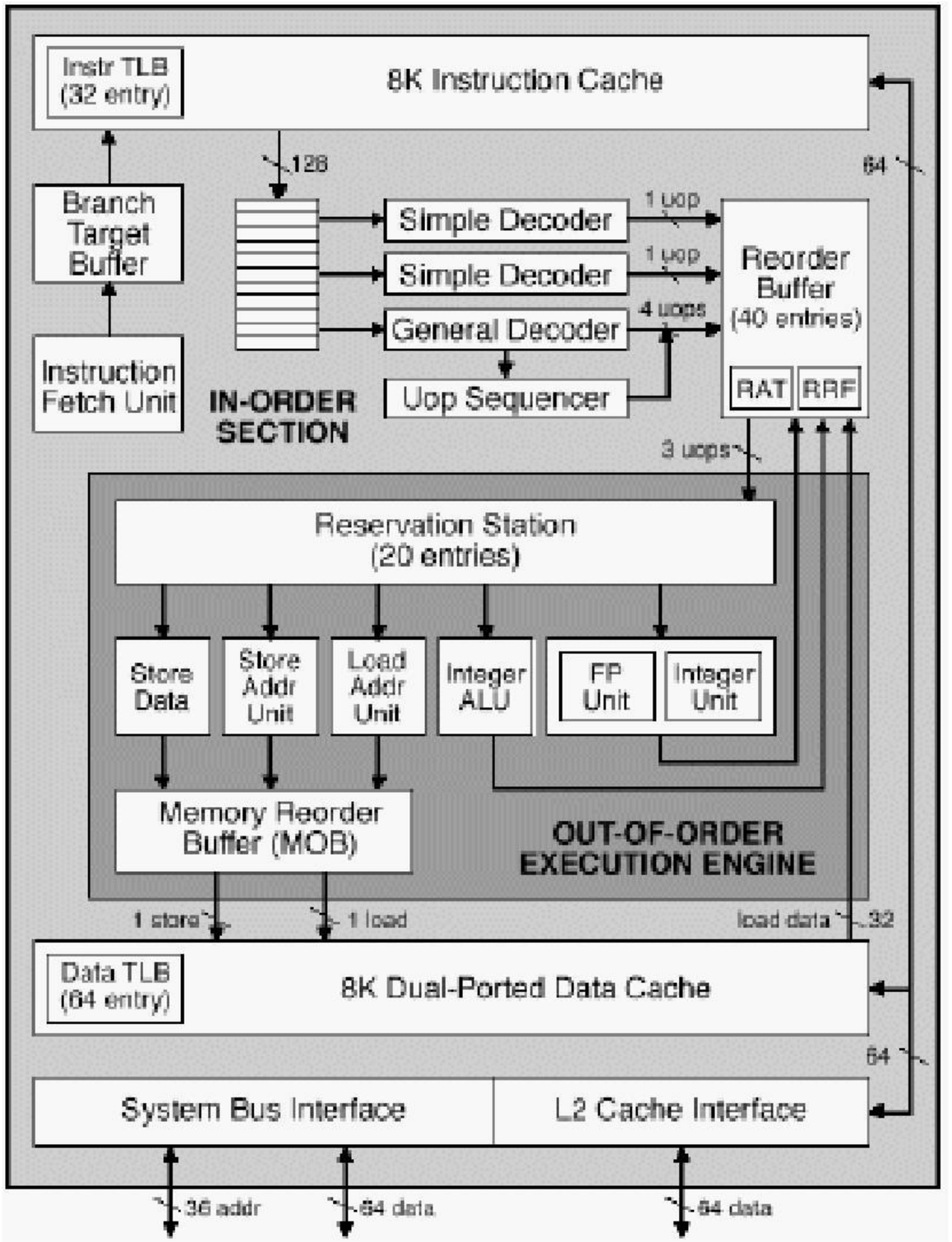
Some Instructions Take > 1 Cycle, but Can be Pipelined

Instruction	Latency	Cycles/Issue
Load / Store	3	1
Integer Multiply	4	1
Integer Divide	36	36
Double/Single FP Multiply	5	2
Double/Single FP Add	3	1
Double/Single FP Divide	38	38

PentiumPro Block Diagram

P6 Microarchitecture

- PentiumPro
- Pentium II
- Pentium III



Microprocessor Report
2/16/95

PentiumPro Operation

Translates instructions dynamically into “Uops”

- 118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine

- Uop executed when
  - Operands available
  - Functional unit available
- Execution controlled by “Reservation Stations”
  - Keeps track of data dependencies between uops
  - Allocates resources

Basic pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

PentiumPro Branch Prediction

Critical to Performance

- 11–15 cycle penalty for misprediction

Branch Target Buffer

- 512 entries
- 4 bits of history
- Adaptive algorithm
  - Can recognize repeated patterns, e.g., alternating taken–not taken

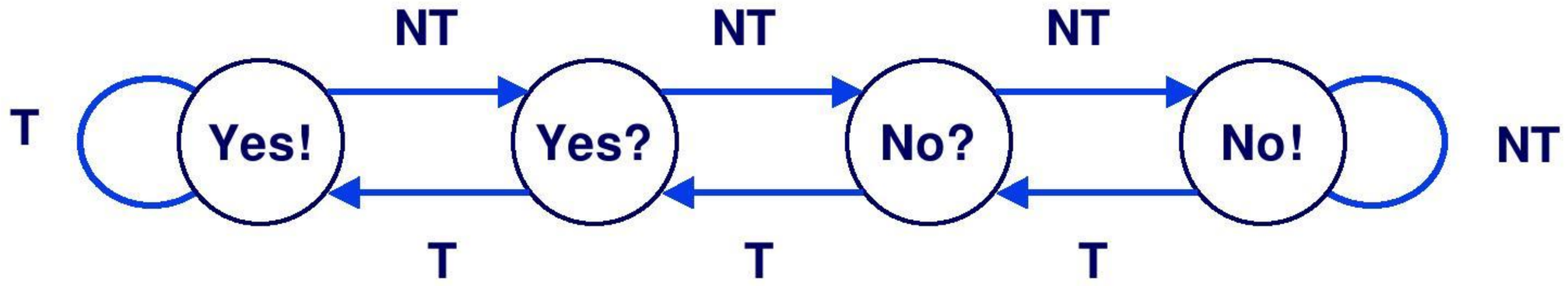
Handling BTB misses

- Detect in cycle 6
- Predict taken for negative offset, not taken for positive
  - Loops vs. conditionals

Example Branch Prediction

Branch History

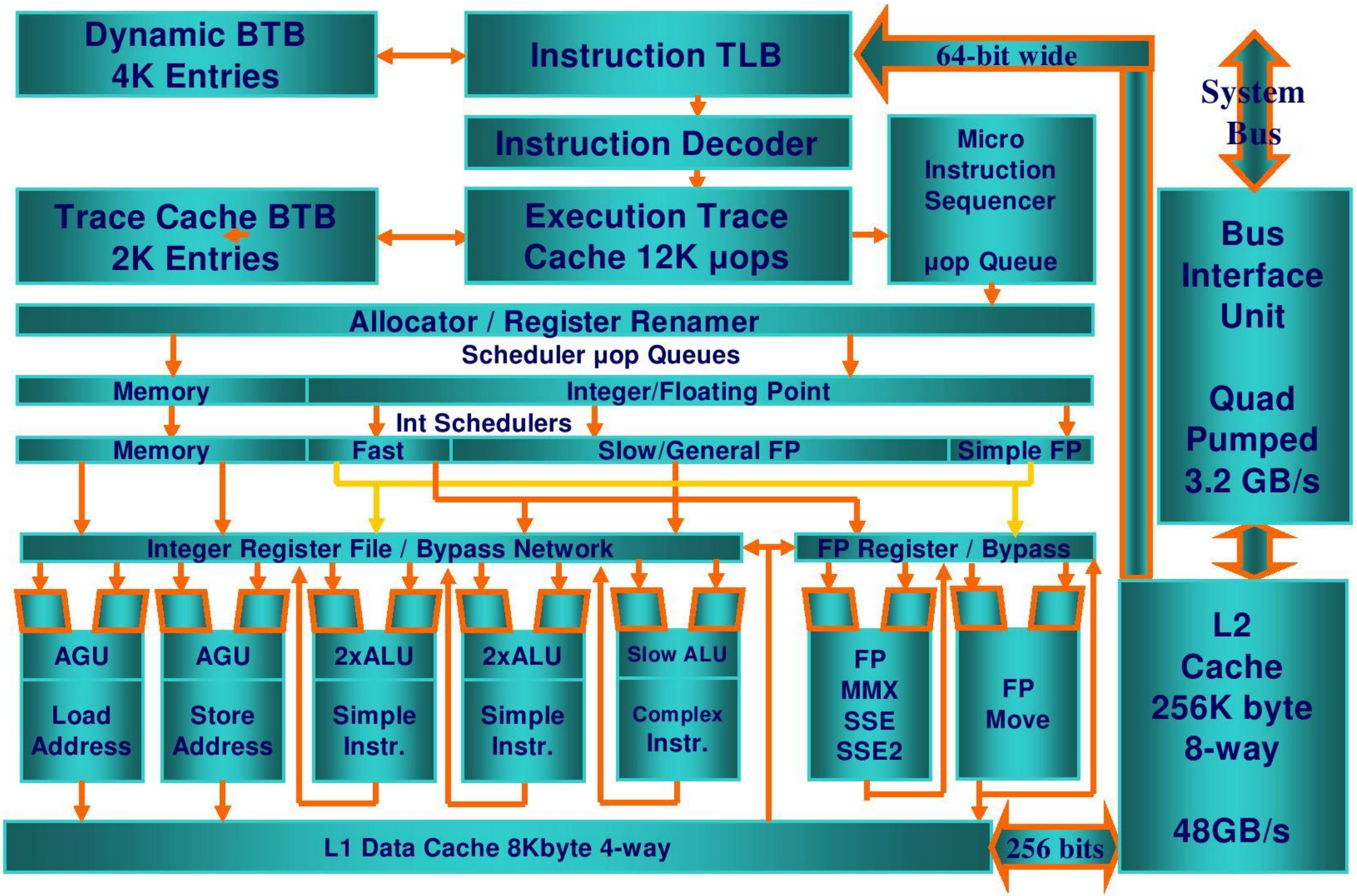
- Encode information about prior history of each individual branch instruction: store as hash table on instr. address
- Predict whether or not branch will be taken



State Machine

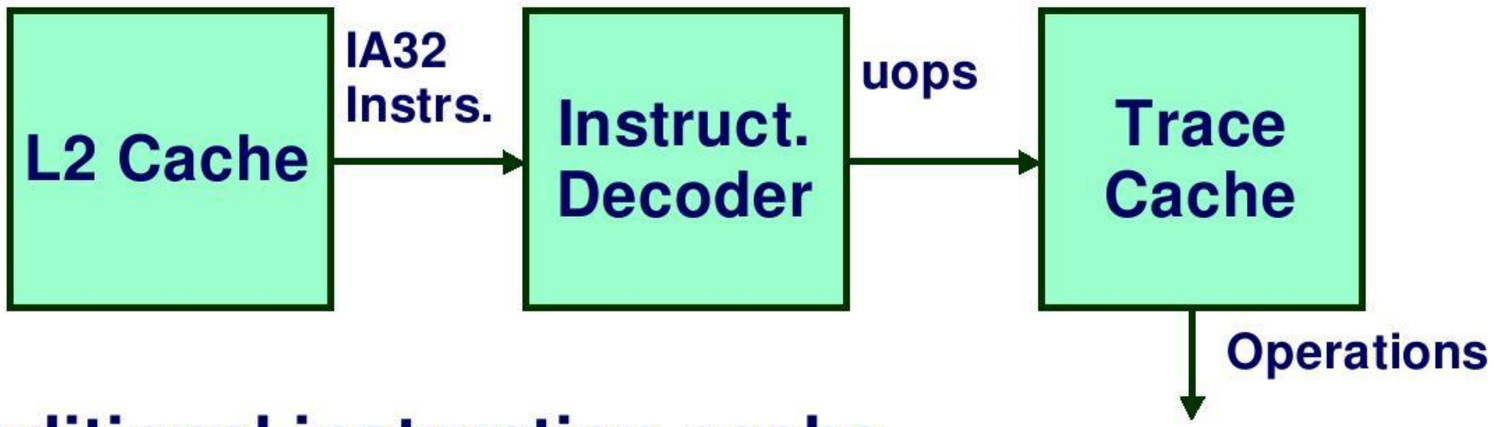
- Each time branch taken, transition to right
- When not taken, transition to left
- Predict branch taken when in state Yes! or Yes?

Pentium® 4 CPU Block Diagram



Pentium 4 Features

Trace Cache



- Replaces traditional instruction cache
- Caches instructions in decoded form

Double-Pumped ALUs

- Simple instructions (add) run at 2X clock rate

Very Deep Pipeline

- 20+ cycle branch penalty: enables very high clock rates
- Slower than Pentium III for a given clock rate

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive