# Organisation und Architektur von Rechnern

Lecture 14

**Instructor:**

Reinhard v. Hanxleden

http://www.informatik.uni-kiel.de/rtsys/teaching/v-sysinf2

*These slides are used with kind permission from the Carnegie Mellon University*

# 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

- **CPI ≈ 1.0**
  - Fetch instruction each clock cycle
  - Effectively process new instruction almost every cycle
    - Although each individual instruction has latency of 5 cycles
- **CPI > 1.0**
  - Sometimes must stall or cancel branches
- **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 due to bubbles

# CPI for PIPE (Cont.)

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

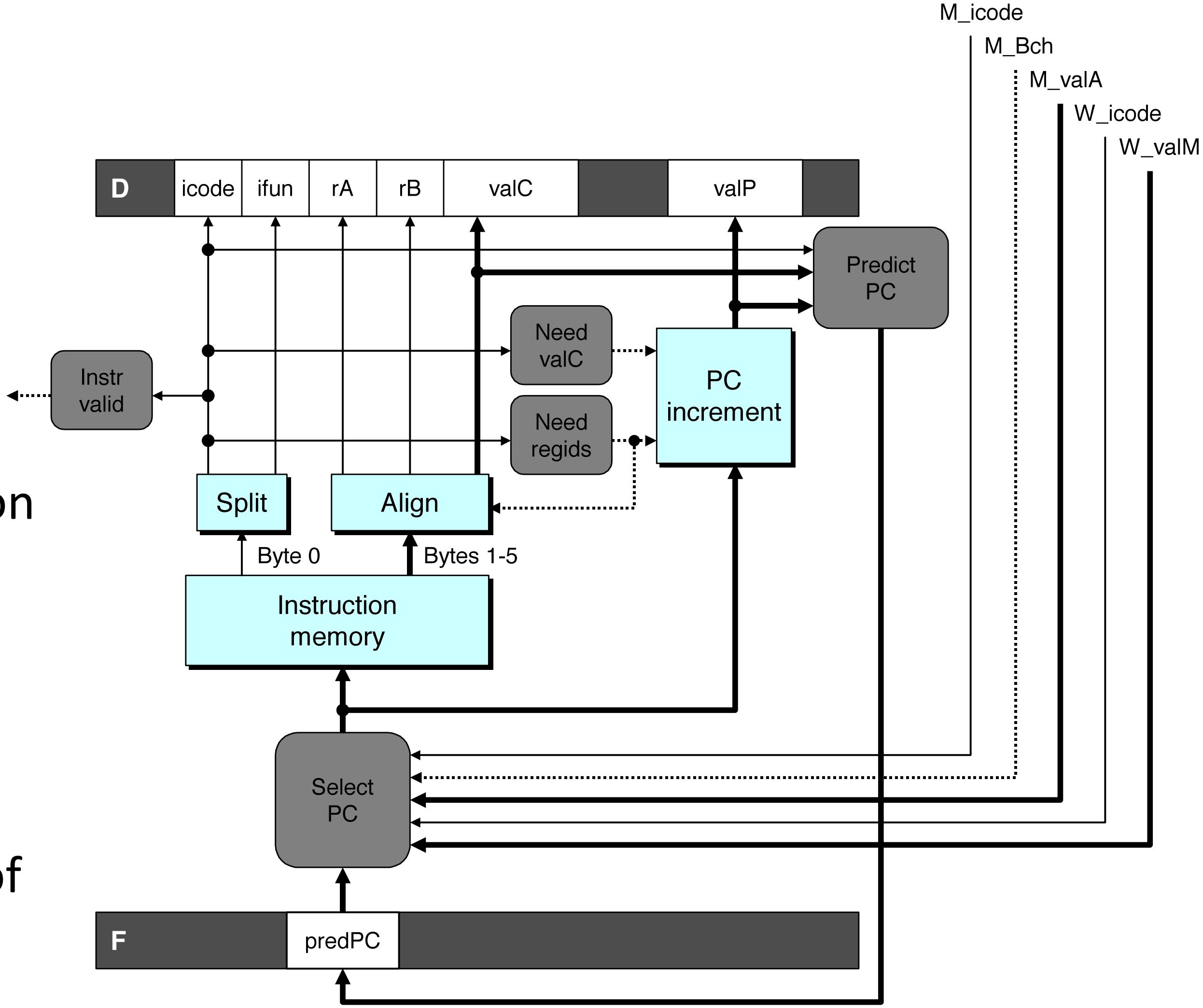
  ⇒ MP = 0.20 * 0.40 * 2 = 0.16
- RP: Penalty due to `ret` instructions
  - Fraction of instructions that are returns — 0.02
  - Number of bubbles injected each time — 3

  ⇒ RP = 0.02 * 3 = 0.06
- Net effect of penalties 0.05 + 0.16 + 0.06 = 0.27

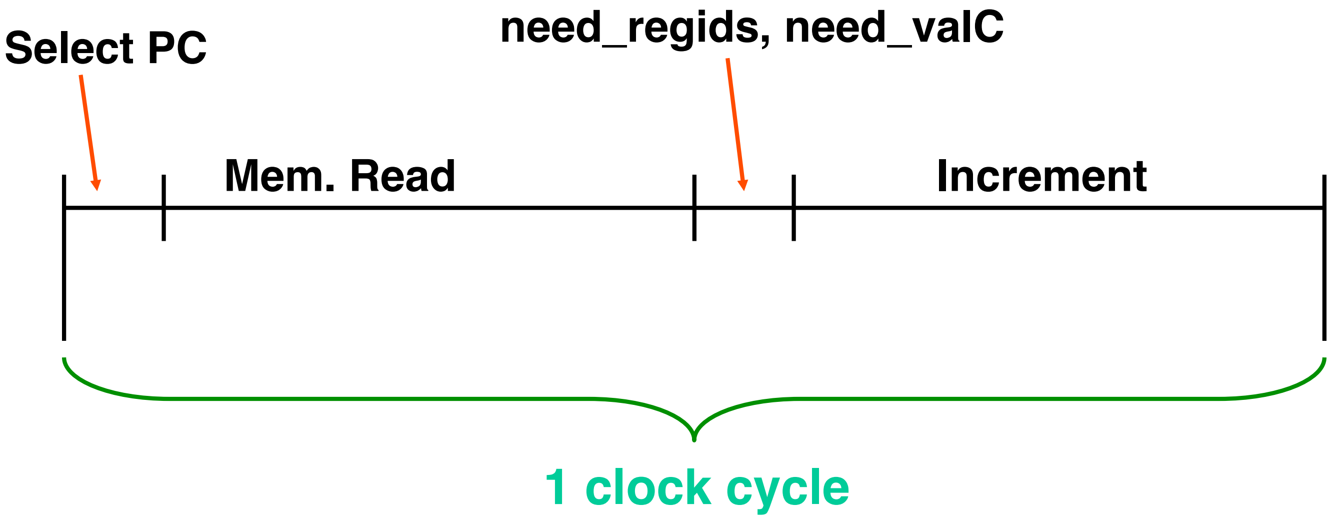
  ⇒ CPI = 1.27 (Not bad!)

# 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

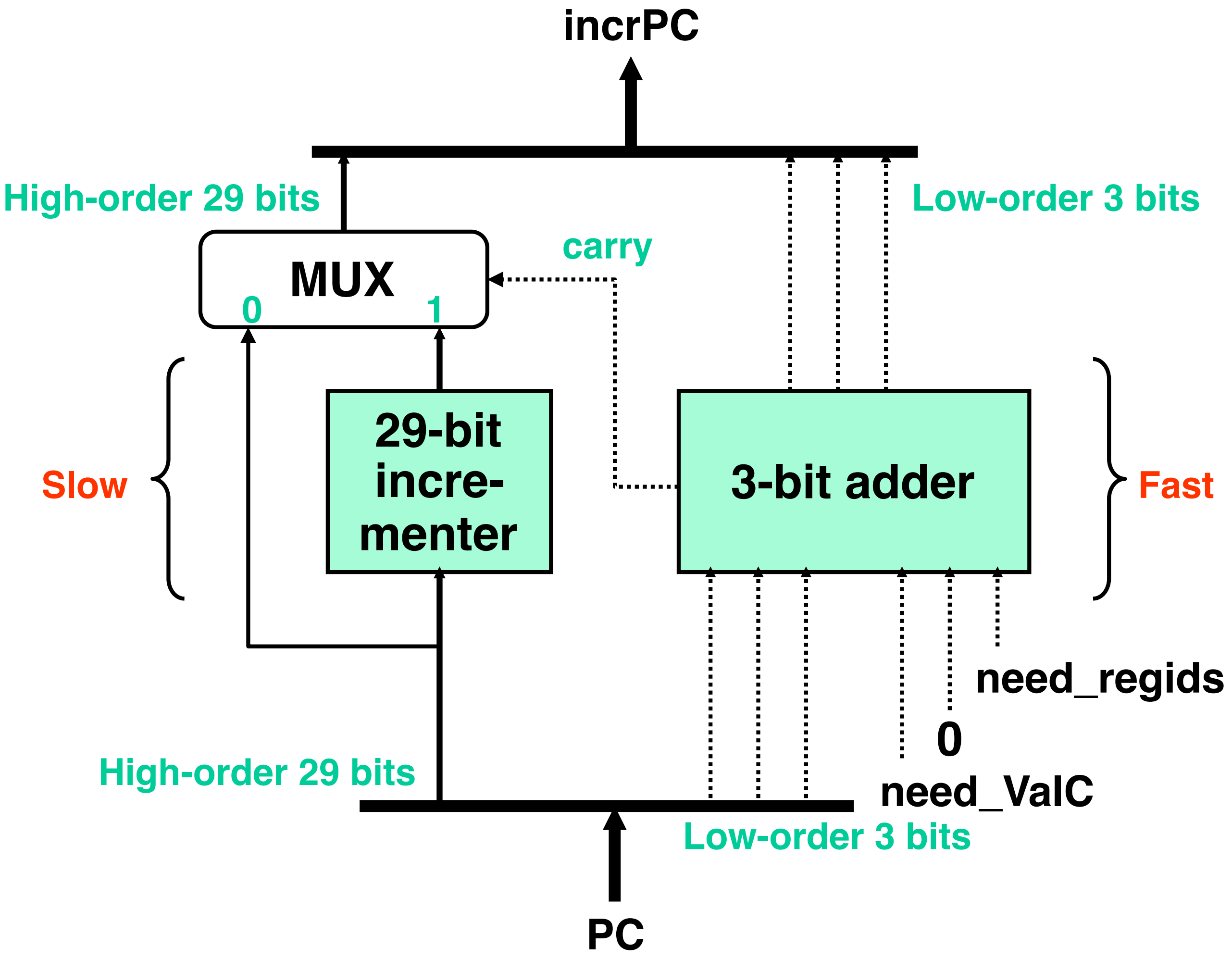


# Standard Fetch Timing

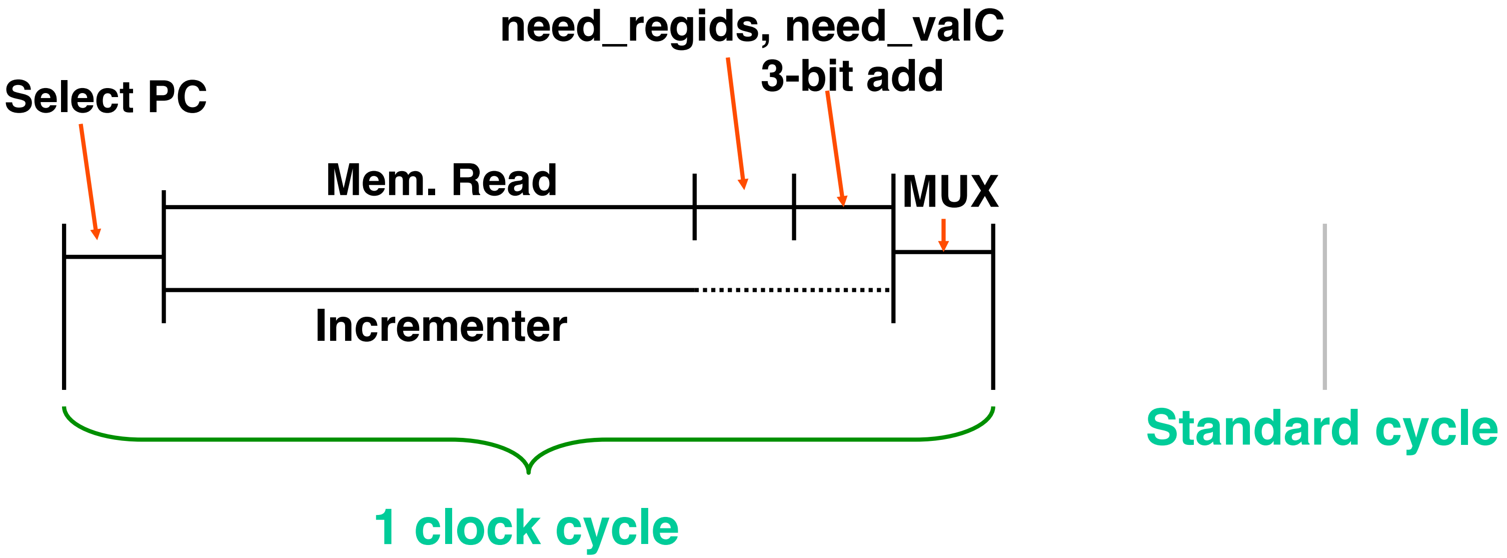


- Must Perform Everything in Sequence
- Can’t compute incremented PC until know how much to increment it by

# A Fast PC Increment Circuit



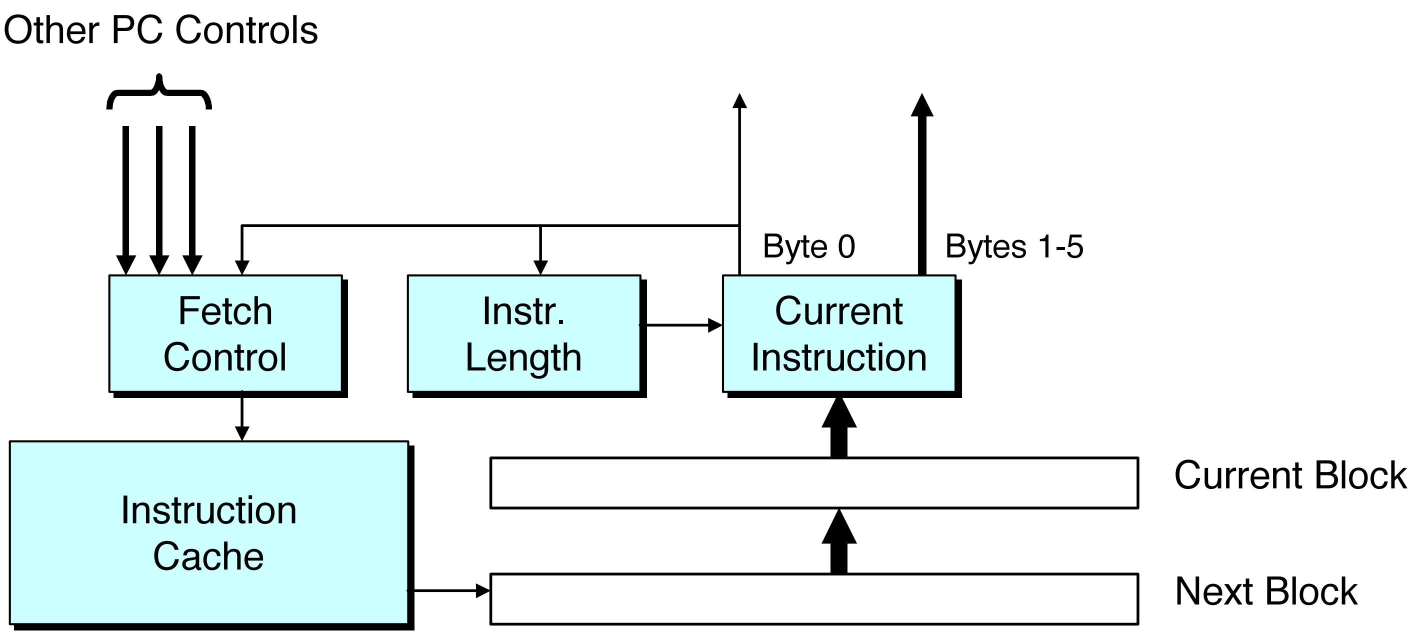
# Modified Fetch Timing



## 29-Bit Incrementer

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

# More Realistic Fetch Logic



- **Fetch Box**
  - Integrated into instruction cache
  - Fetches entire cache block (16 or 32 bytes)
  - Selects current instruction from current block
  - Works ahead to fetch next block
    - As reaches end of current block
    - At branch target

# 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 exception handler
    - Like an unexpected procedure call

# Exception Examples

- **Detect in Fetch Stage**

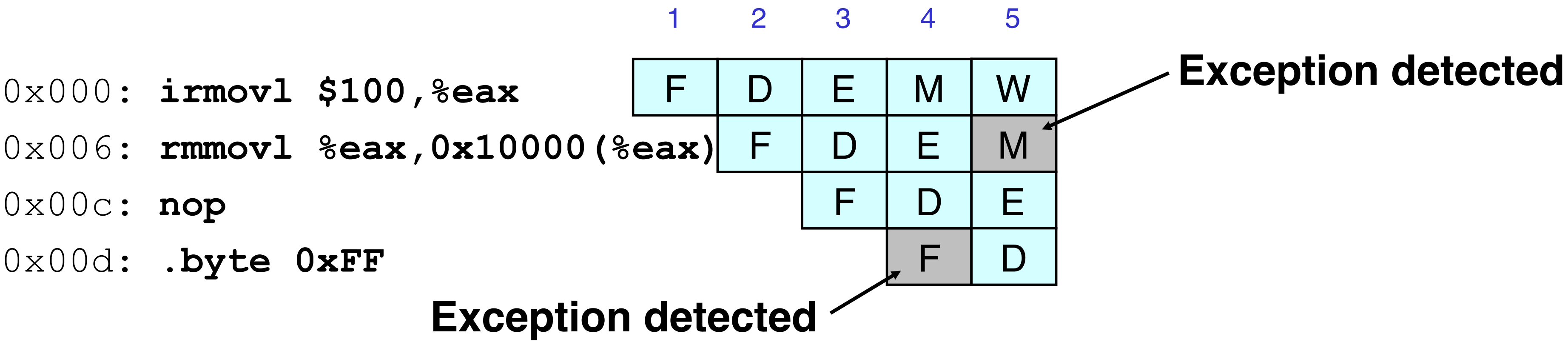
```
jmp $-1                          # Invalid jump target

.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
```



| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 0x000: irmovl $100,%eax | F | D | E | M | W |
| 0x006: rmmovl %eax,0x10000(%eax) | | F | D | E | M |
| 0x00c: nop | | | F | D | E |
| 0x00d: .byte 0xFF | | | | F | D |

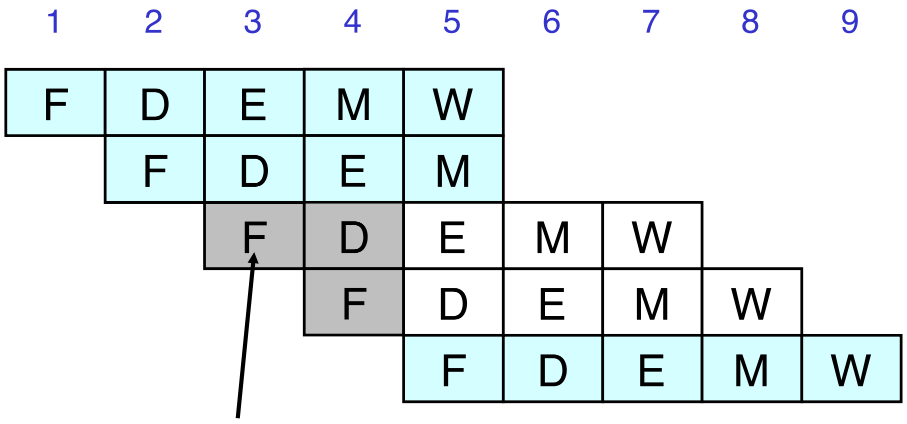
Exception detected

Exception detected

- **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
```

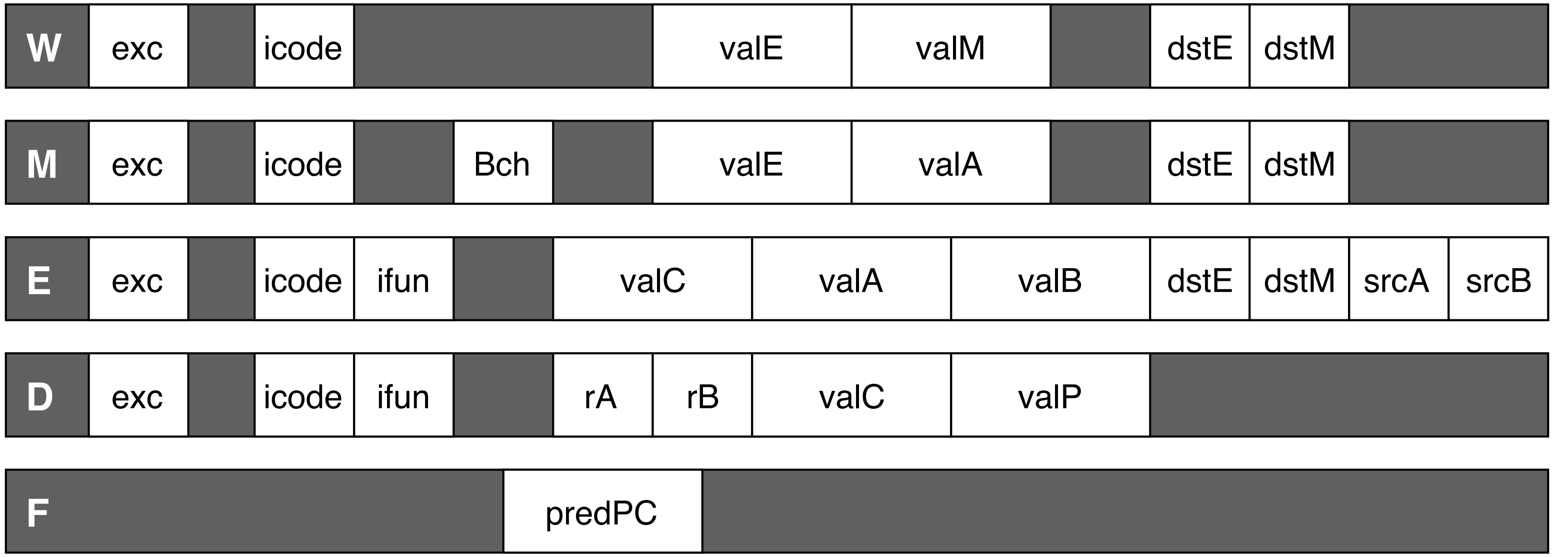


| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `0x000:    xorl %eax,%eax` | F | D | E | M | W | | | | |
| `0x002:    jne t` | | F | D | E | M | | | | |
| `0x014: t: .byte 0xFF` | | | F | D | E | M | W | | |
| `0x???: (I’m lost!)` | | | | F | D | E | M | W | |
| `0x007:    irmovl $1,%eax` | | | | | F | D | E | M | W |

Exception detected

- **Desired Behavior**
  - No exception should occur

# Maintaining Exception Ordering



W exc icode valE valM dstE dstM

M exc icode Bch valE valA dstE dstM

E exc icode ifun valC valA valB dstE dstM srcA srcB

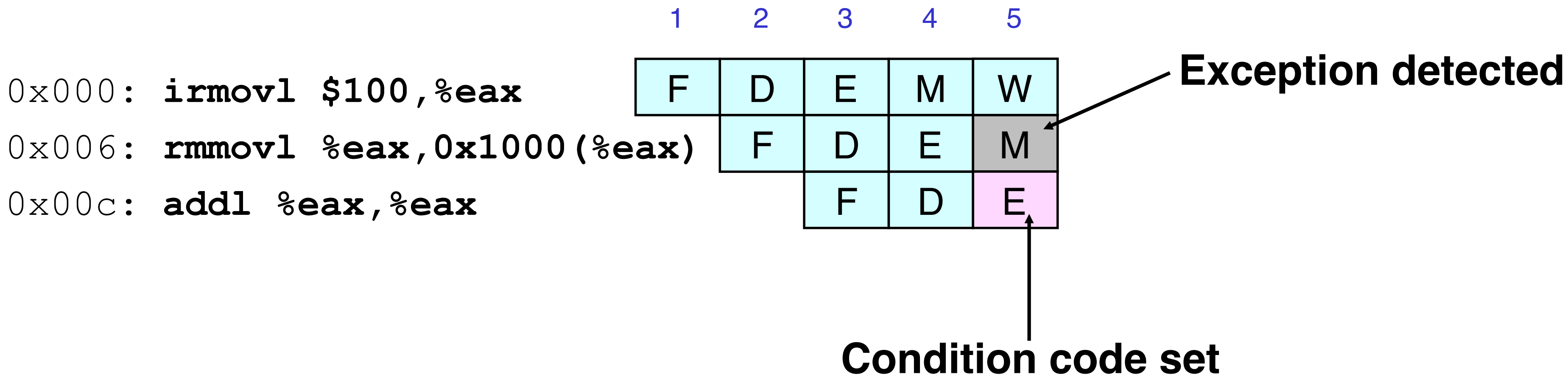
D exc icode ifun rA rB valC valP

F predPC

- 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
```



| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 0x000: irmovl $100,%eax | F | D | E | M | W |
| 0x006: rmmovl %eax,0x1000(%eax) | | F | D | E | M |
| 0x00c: addl %eax,%eax | | | F | D | E |

- **Desired Behavior**
  - rmmovl should cause exception
  - No following instruction should have any effect

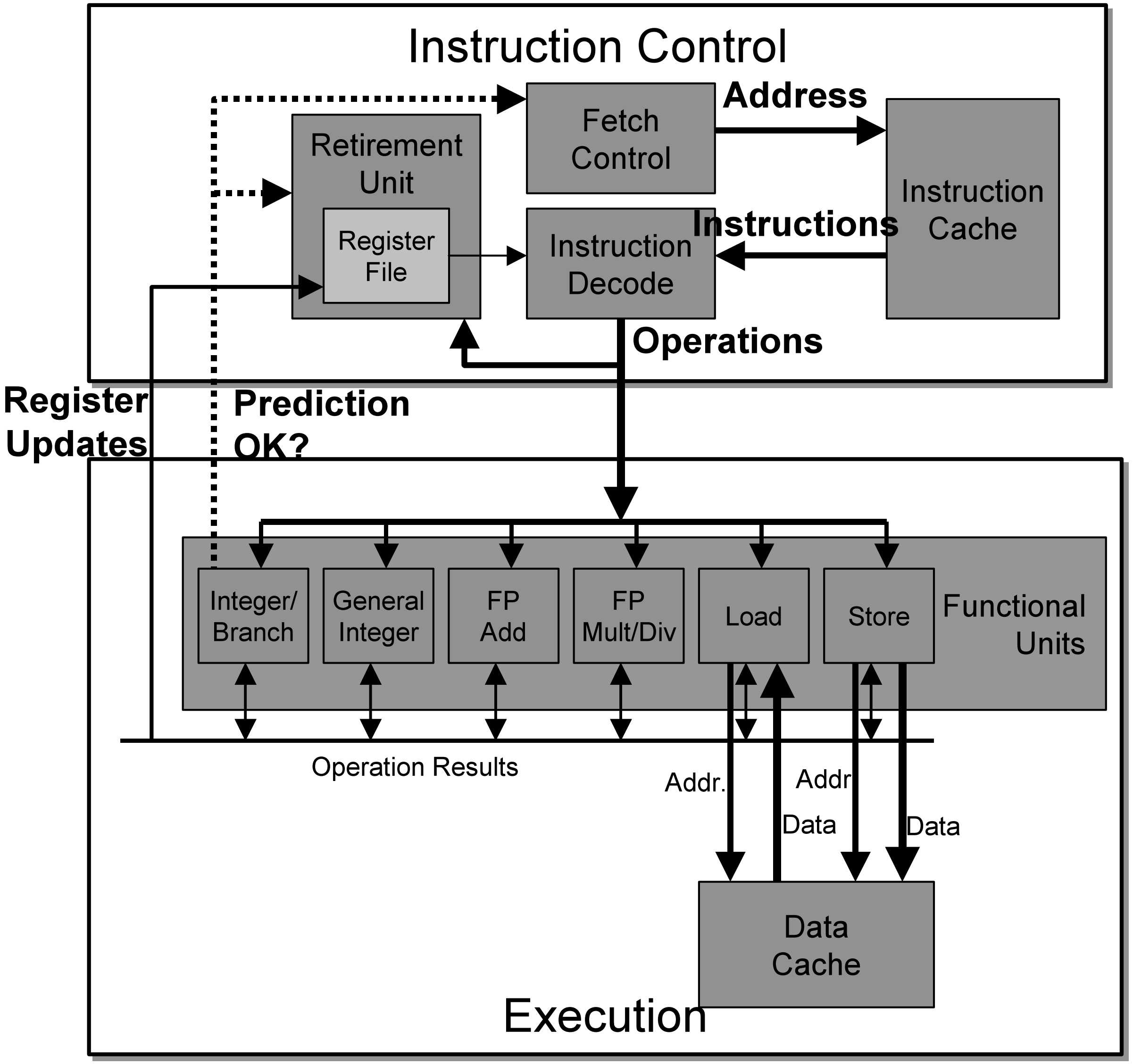
# Avoiding Side Effects

- **Presence of Exception Should Disable State Update**
  - When detect exception 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**
  - Hardwired into the design of the PIPE simulator
  - You have no control over this

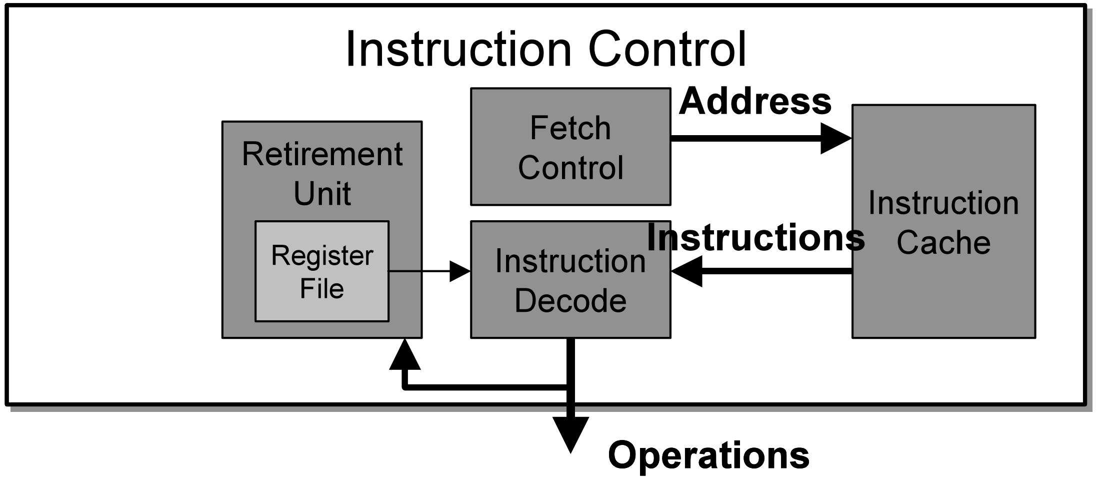
# Rest of Exception Handling

- **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**
  - Haven’t tried it yet!

# Modern CPU Design

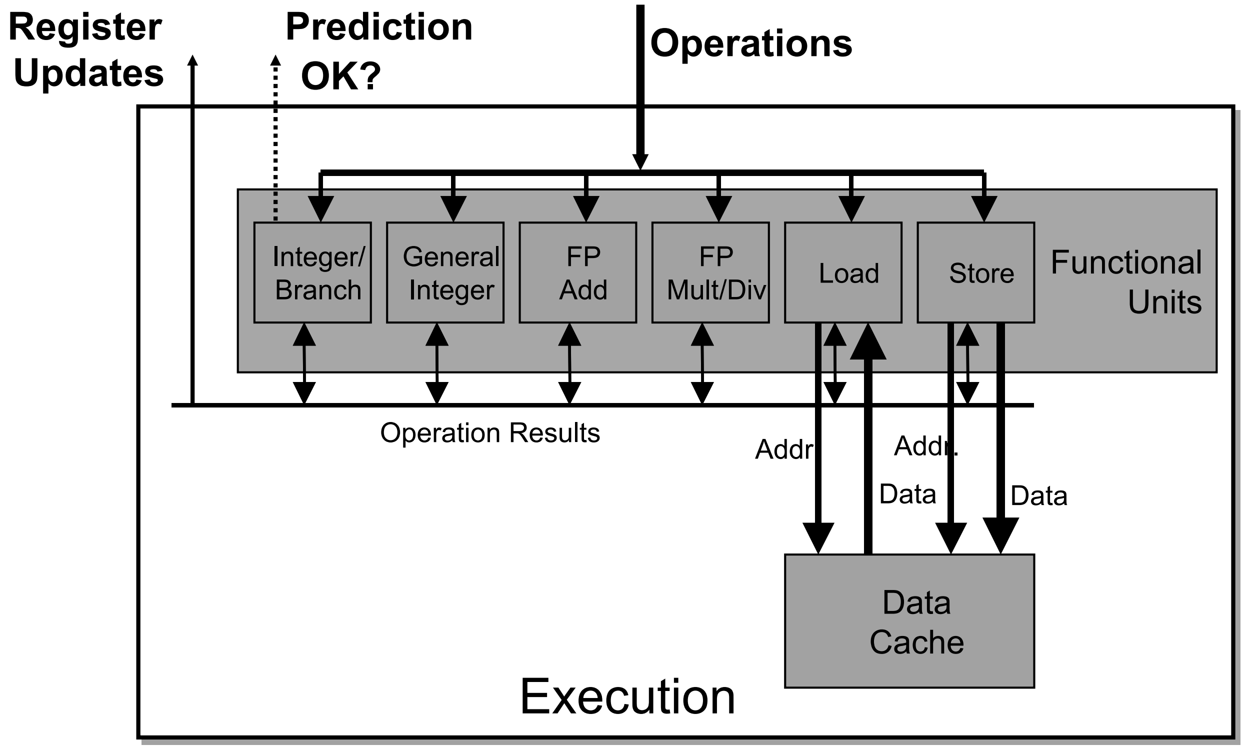


# Instruction Control



- **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 performed as soon as operands available
  - Not necessarily in program order
  - Within limits of functional units
- Control logic
  - Ensures behavior equivalent to sequential program execution

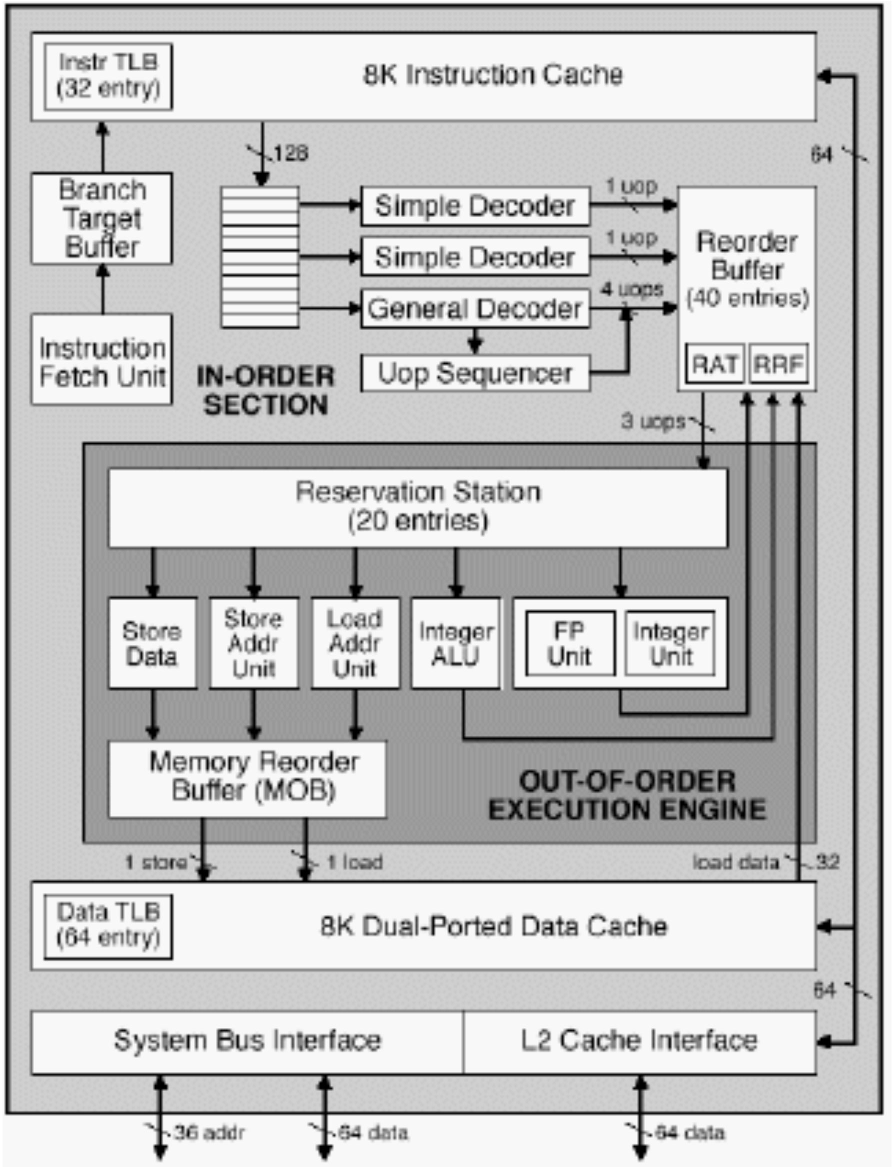
# CPU Capabilities of Pentium III

- **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

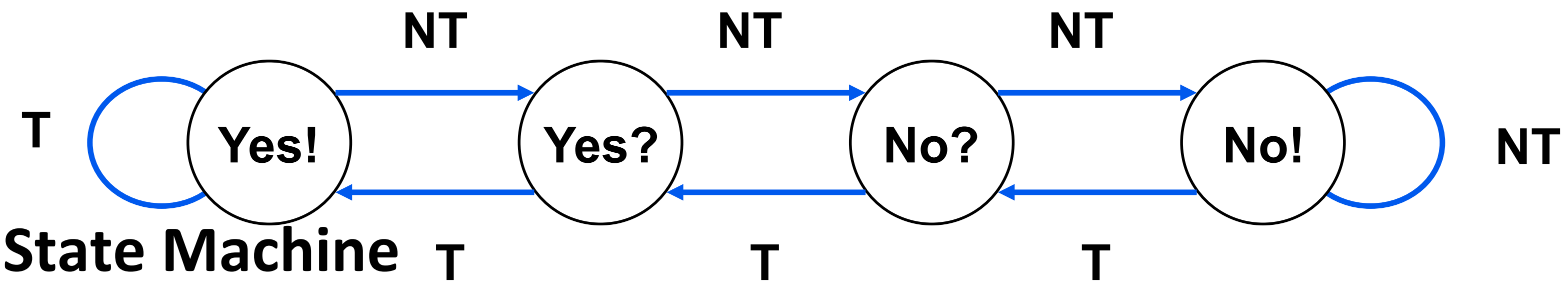
- **Translates instructions dynamically into “Uops”**
  - 118 bits wide
  - Holds operation, two sources, and destination
- **Executes Uops with “Out of Order” engine**
  - Uop executed when
    - Operands available
    - Functional unit available
  - Execution controlled by “Reservation Stations”
    - Keeps track of data dependencies between uops
    - Allocates resources

# 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

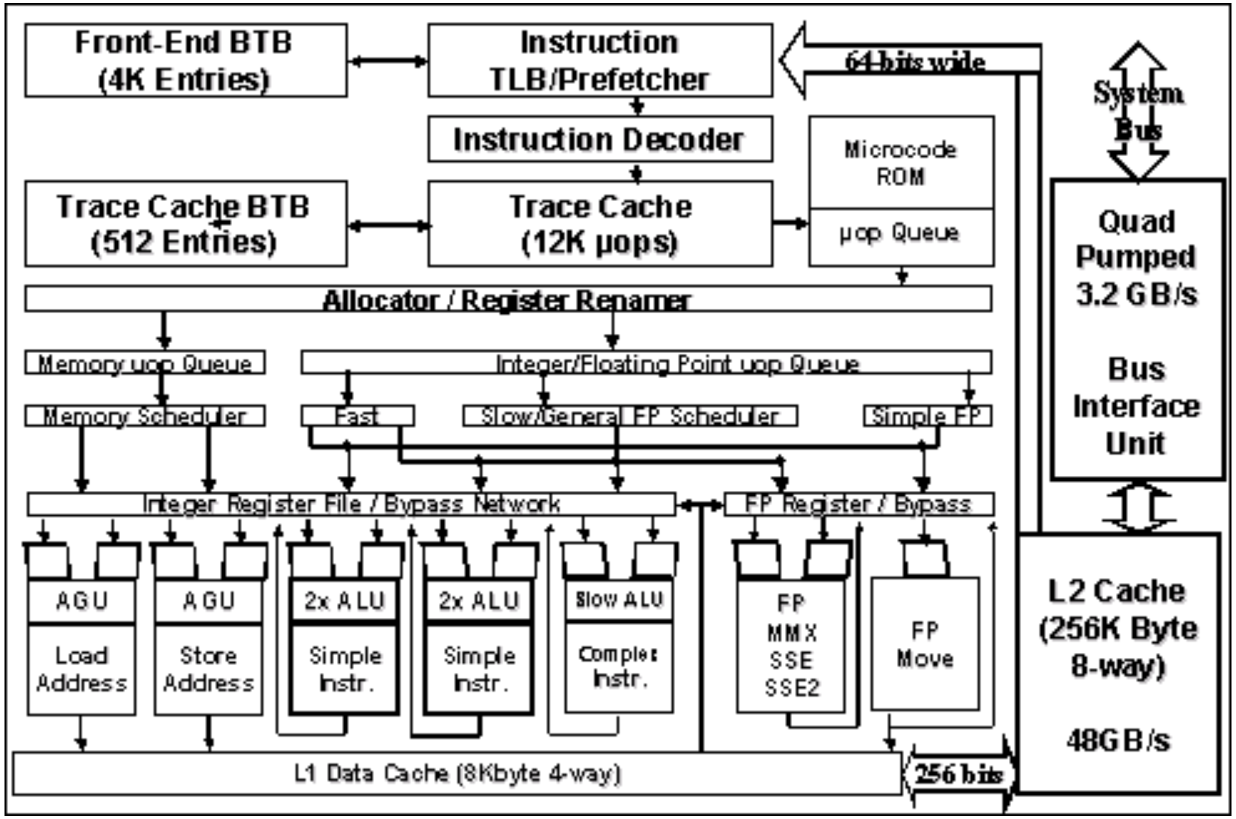
# Example Branch Prediction

- **Branch History**
  - Encode information about prior history of branch instructions
  - 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 Block Diagram

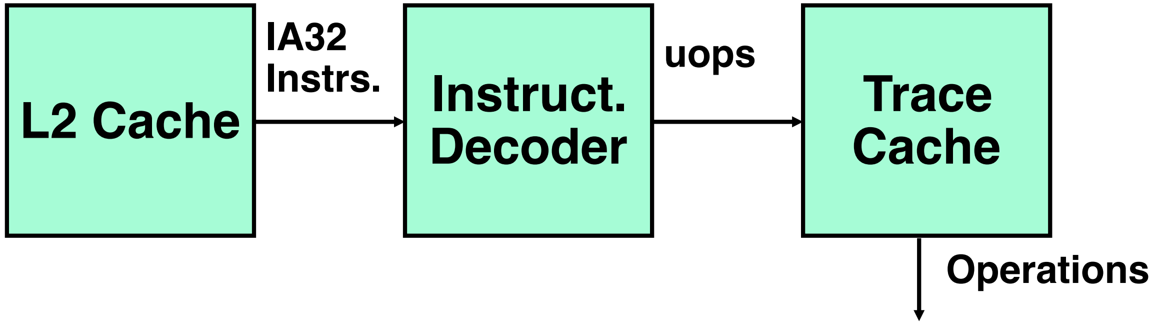


**Intel Tech. Journal Q1, 2001**

- Next generation microarchitecture

# Pentium 4 Features

- **Trace Cache**



  - Replaces traditional instruction cache
  - Caches instructions in decoded form
  - Reduces required rate for instruction decoder
- **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

# Processor Summary

- **Design Technique**
  - Create uniform framework for all instructions
    - Want to share hardware among instructions
  - Connect standard logic blocks with bits of control logic
- **Operation**
  - State held in memories and clocked registers
  - Computation done by combinational logic
  - Clocking of registers/memories sufficient to control overall behavior
- **Enhancing Performance**
  - Pipelining increases throughput and improves resource utilization
  - Must make sure maintains ISA behavior