The Underlying Architecture

PSU CS 532

Prof. Karen L. Karavanic

Portland State University

Acknowledgments

- This presentation includes or is motivated by materials developed by others:
  - 15-213: Introduction to Computer Systems, 2010
    - Randy Bryant and Dave O'Hallaron
  - Kathy Yelick, UC-Berkeley
  - Andrew S. Tanenbaum, *Modern Operating Systems*
  - Remzi and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*
  - Jonathan Walpole, Bruce Irvin Portland State University

The Single Core Era #5: Key Architecture Advances

- Instruction Level Parallelism (ILP)
- Pipelining
- Branch Prediction
- Multiple Instruction Issue
- The Memory Hierarchy

Pipelining

The Insight

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions
- Design hardware so that a different instruction can be at each step concurrently
- 1. Instr 1 at stage 1
- 2. Instr 1 at stage 2, Instr 2 at stage 1
- 3. Instr 1 at stage 3, Instr 2 at stage 2, Instr 3 at stage 1
- …

Real-World Pipelines: Car Washes

Sequential

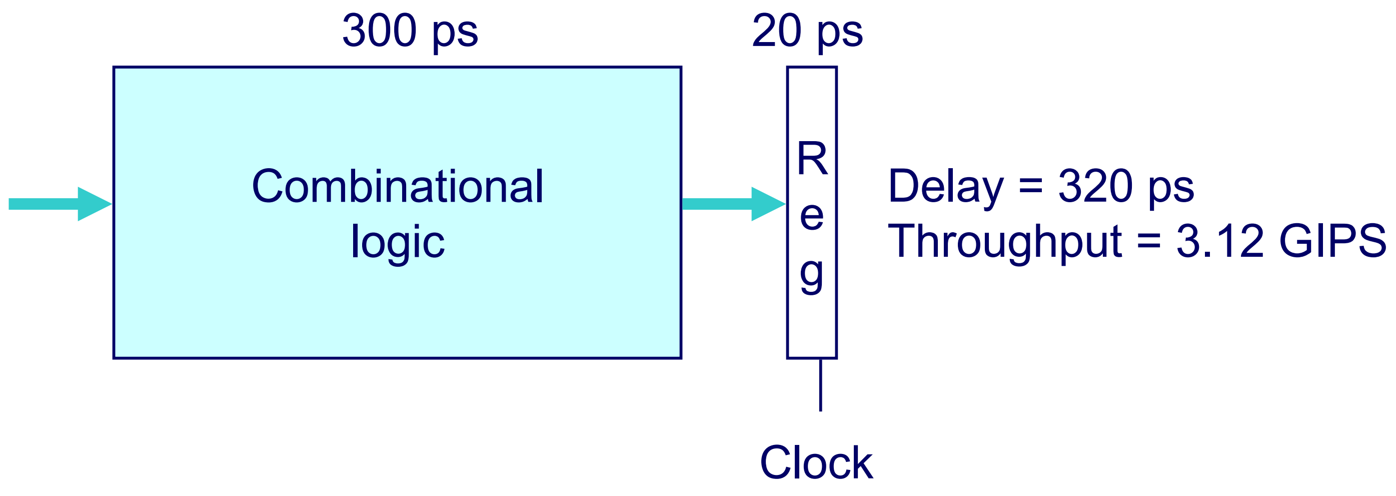
Parallel

Pipelined

Idea

- Divide process into independent stages
- Move each car through stages in sequence
- At any given time, multiple cars being processed

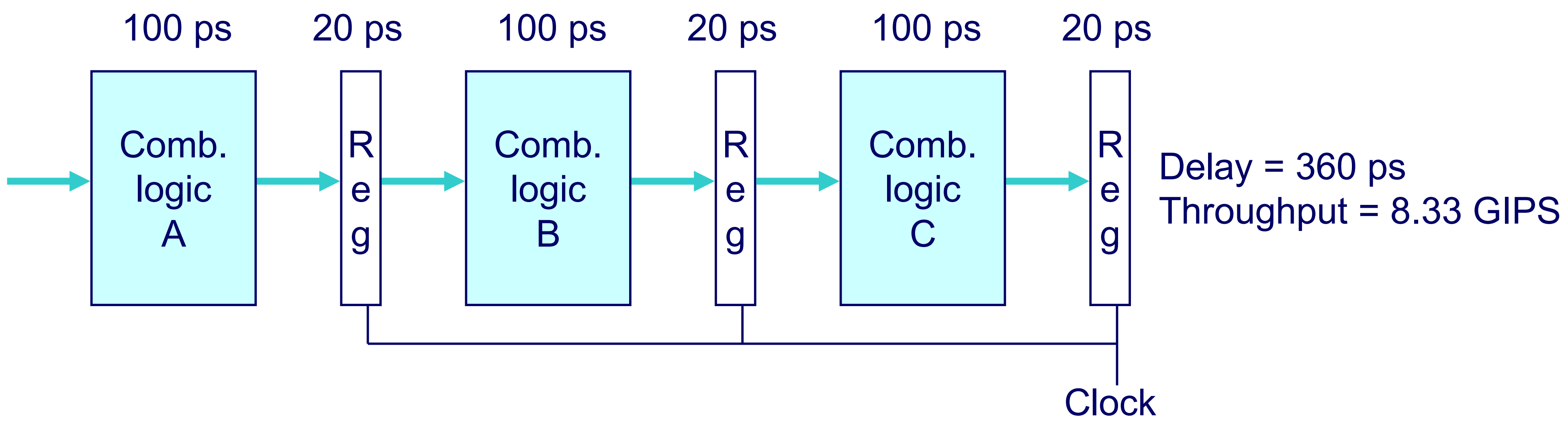
Computational Example



System

- **Computation requires total of 300 picoseconds**
- **Additional 20 picoseconds to save result in register**
- **Must have clock cycle of at least 320 ps**

3-Way Pipelined Version

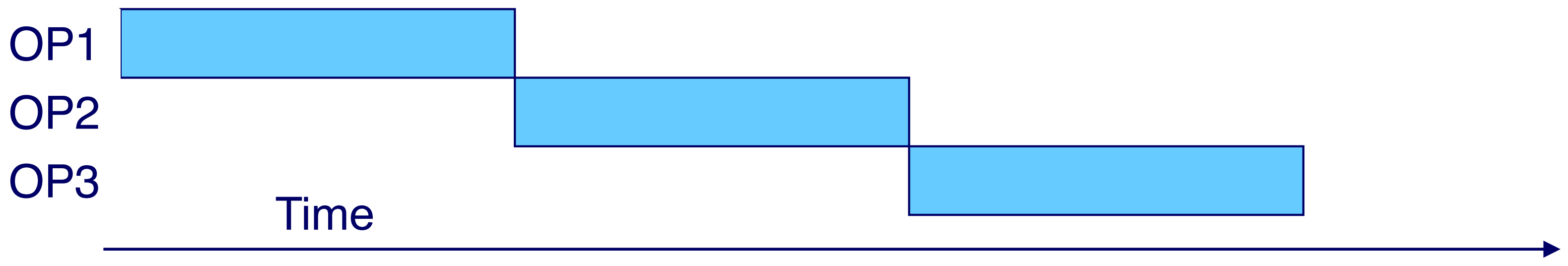


System

- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous one passes through stage A.
  - Begin new operation every 120 ps
- Overall latency increases
  - 360 ps from start to finish

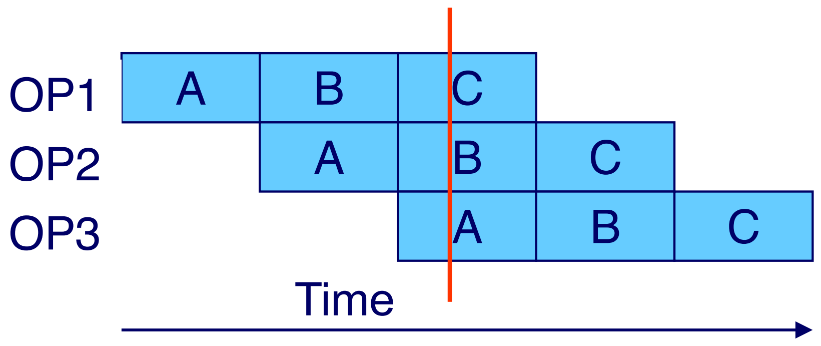
Pipeline Diagrams

Unpipelined



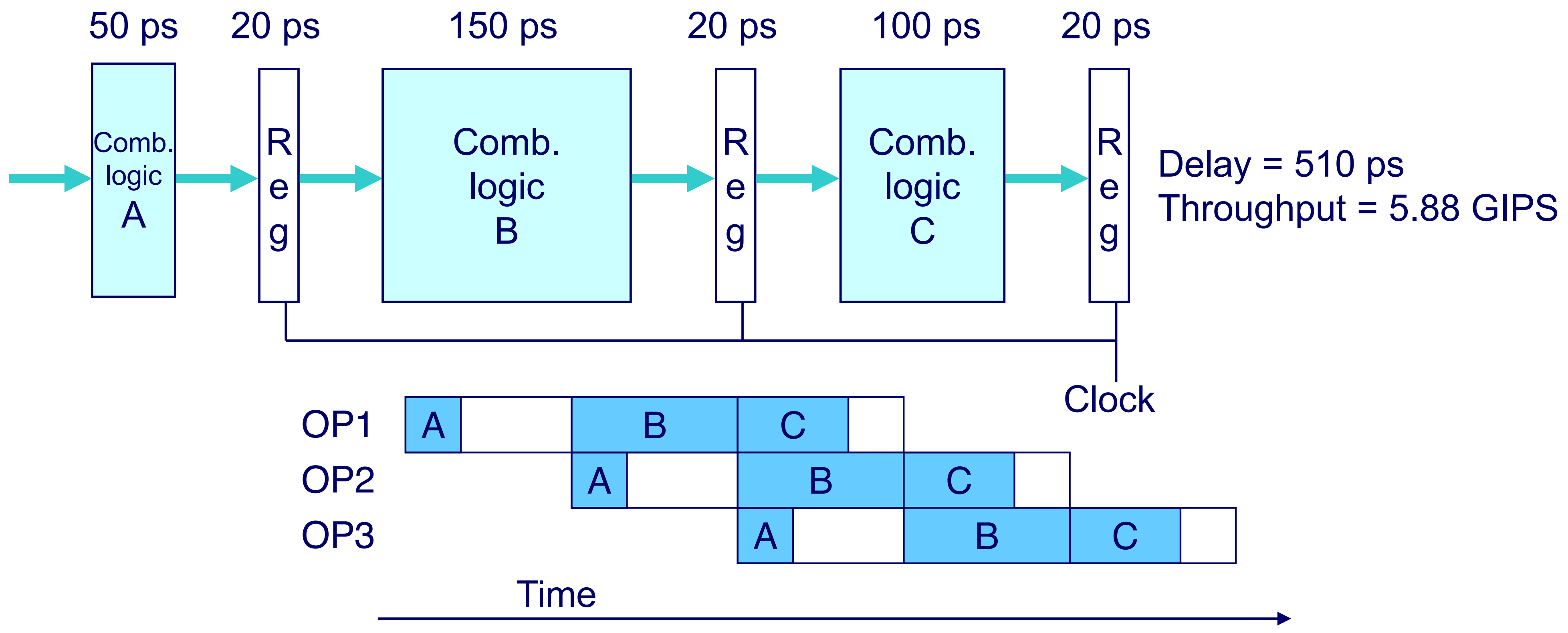
- Cannot start new operation until previous one completes

3-Way Pipelined



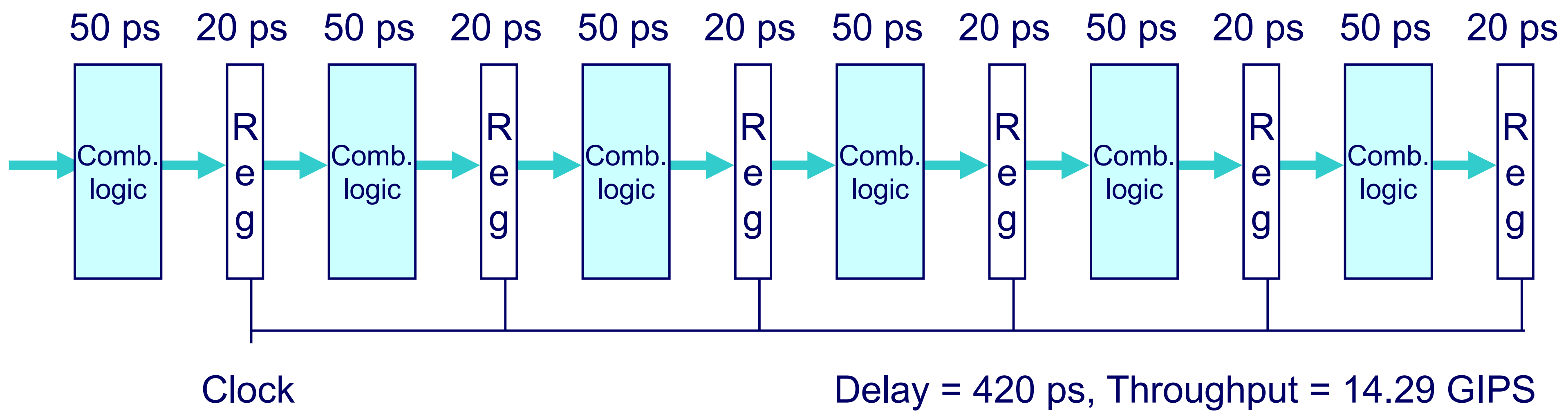
- Up to 3 operations in process simultaneously

Limitations: Nonuniform Delays



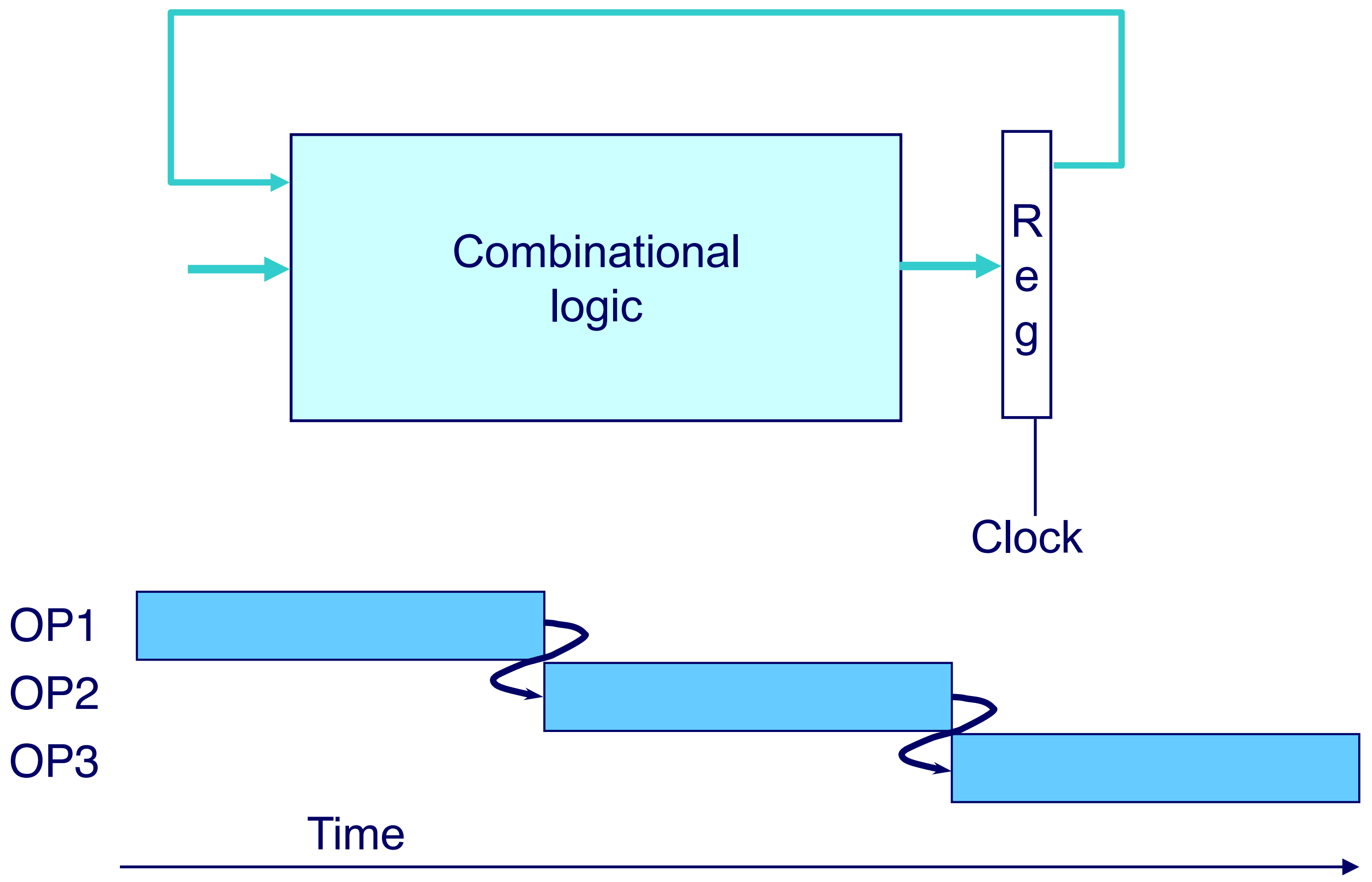
- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Limitations: Register Overhead



- **As try to deepen pipeline, overhead of loading registers becomes more significant**
- **Percentage of clock cycle spent loading register:**
  - **1-stage pipeline: 6.25%**
  - **3-stage pipeline: 16.67%**
  - **6-stage pipeline: 28.57%**
- **High speeds of modern processor designs obtained through very deep pipelining**

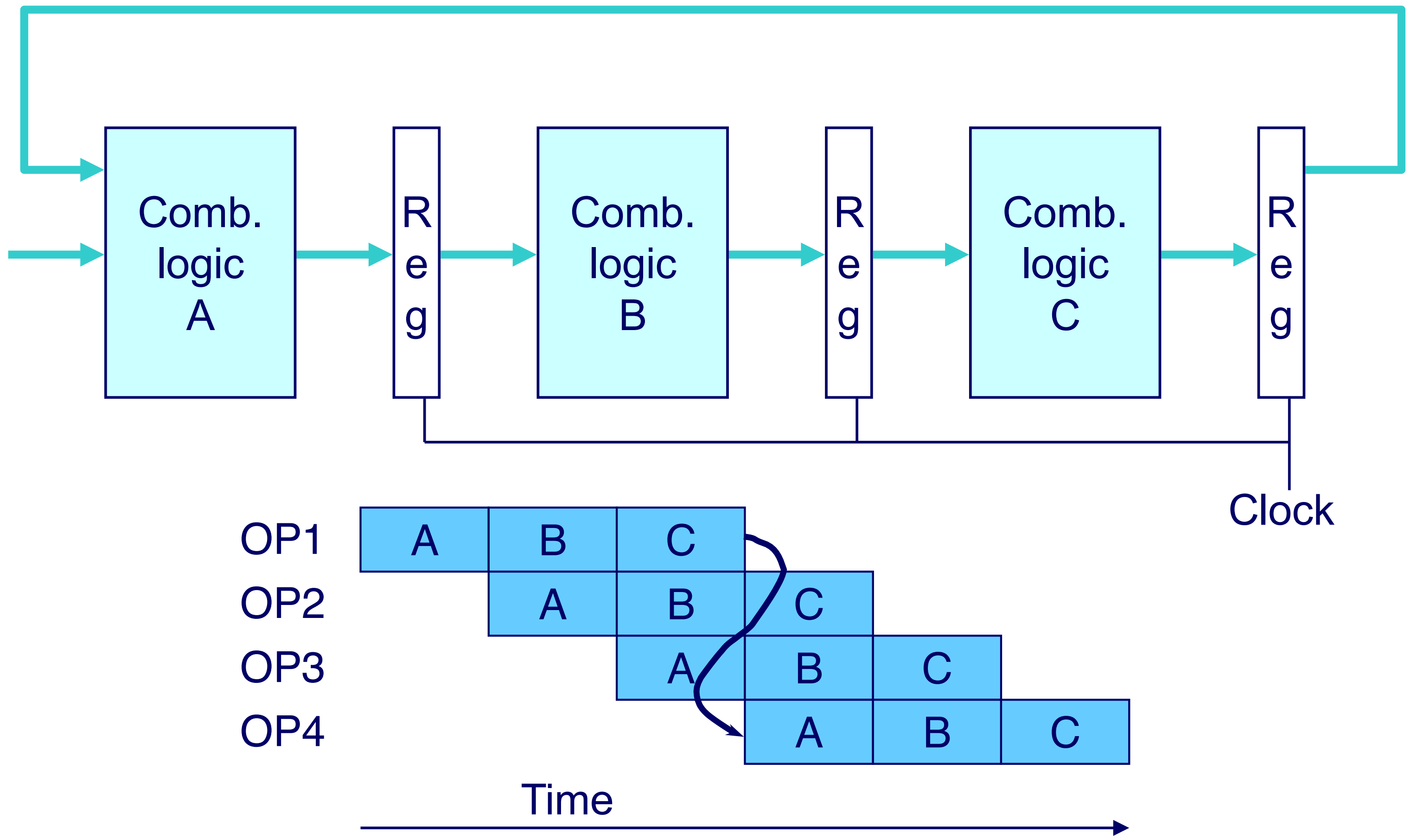
Pipelining Challenges: Data Dependencies



System

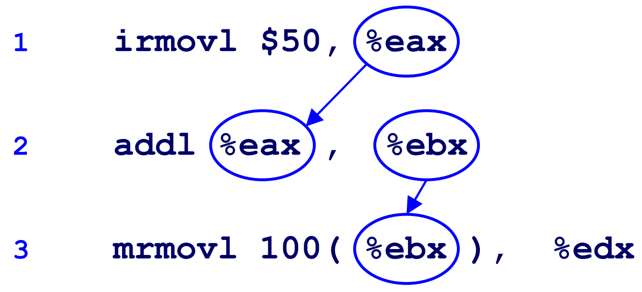
- Each operation depends on result from preceding one

Pipelining Challenges: Data Hazards



- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

Data Dependencies in Processors



```
irmovl $50, %eax
addl %eax , %ebx
mrmovl 100( %ebx ),  %edx
```

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

Pipeline Stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

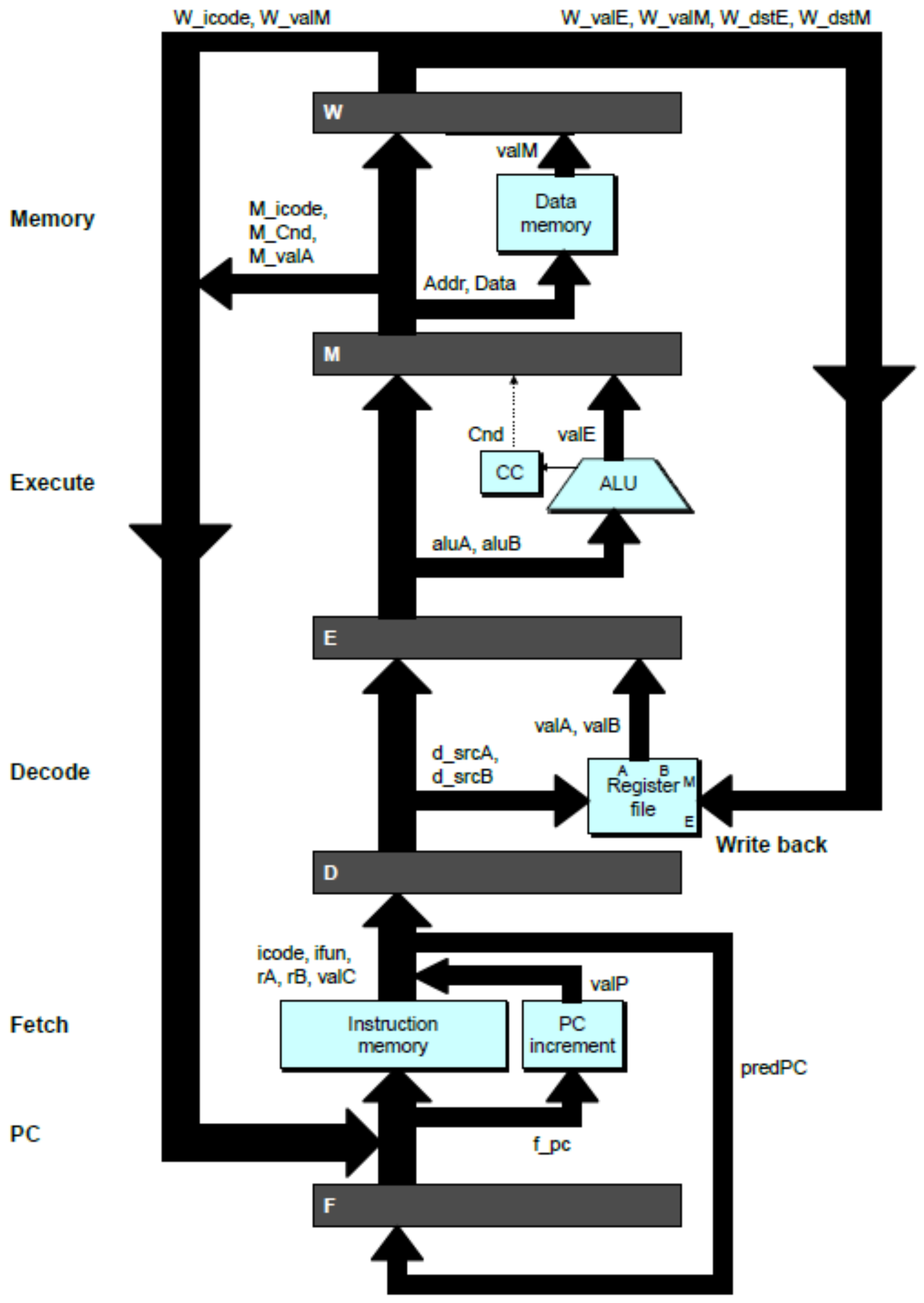
- Operate ALU

Memory

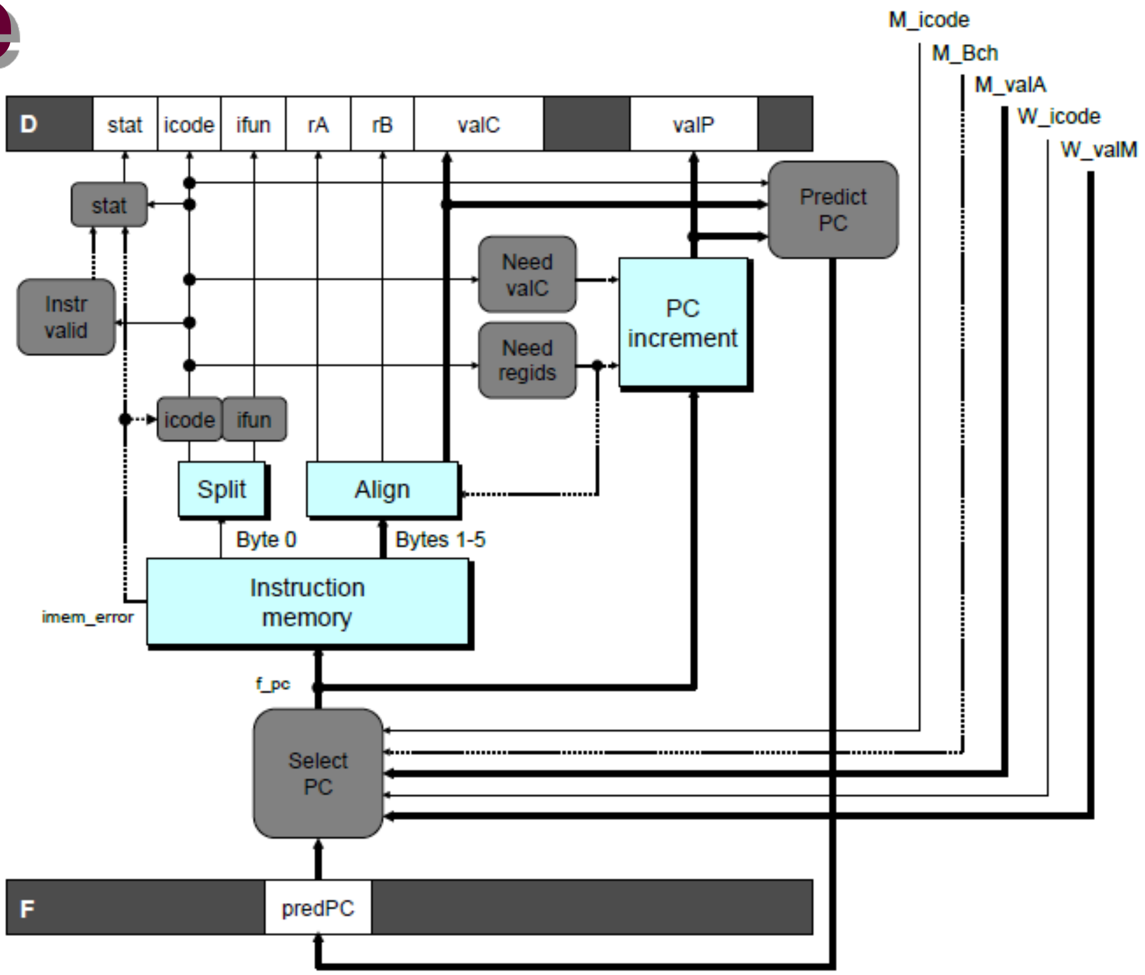
- Read or write data memory

Write Back

- Update register file



Predicting the PC



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

Example: Prediction Strategy

Instructions that Don’t Transfer Control

- Predict next PC to be valP
- Always reliable

Call and Unconditional Jumps

- Predict next PC to be valC (destination)
- Always reliable

Conditional Jumps

- Predict next PC to be valC (destination)
- Only correct if branch is taken
  - Typically right 60% of time

Return Instruction

- Don’t try to predict

Pipeline Summary

Concept

- Break instruction execution into 5 stages
- Run instructions through in pipelined mode

Limitations

- Can’t handle dependencies between instructions when instructions follow too closely
- Data dependencies
  - One instruction writes register, later one reads it
- Control dependency
  - Instruction sets PC in way that pipeline did not predict correctly
  - Mispredicted branch and return

Superscalar Processor

- Definition: A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- Benefit: without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have

- Most CPUs since about 1998 are superscalar.
- Intel: since Pentium Pro

Superscaler example: Nehalem CPU

- **Multiple instructions can execute in parallel**

  1 load, with address computation

  1 store, with address computation

  2 simple integer (one may be branch)

  1 complex integer (multiply/divide)

  1 FP Multiply

  1 FP Add

- **Some instructions take > 1 cycle, but can be pipelined**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**11--21**	**11--21**
Single/Double FP Multiply	4/5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**10--23**	**10--23**

Loop Unrolling

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Latency Bound	1.0	3.0	3.0	5.0

- **Helps integer multiply**
  - below latency bound
  - Compiler does clever optimization
- **Others don't improve. *Why?***
  - Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```

What About Branches?

- **Challenge**
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy

```
80489f3:  movl    $0x1,%ecx
80489f8:  xorl    %edx,%edx
80489fa:  cmpl    %esi,%edx
80489fc:  jnl     8048a25
80489fe:  movl    %esi,%esi
8048a00:  imull   (%eax,%edx,4),%ecx
```

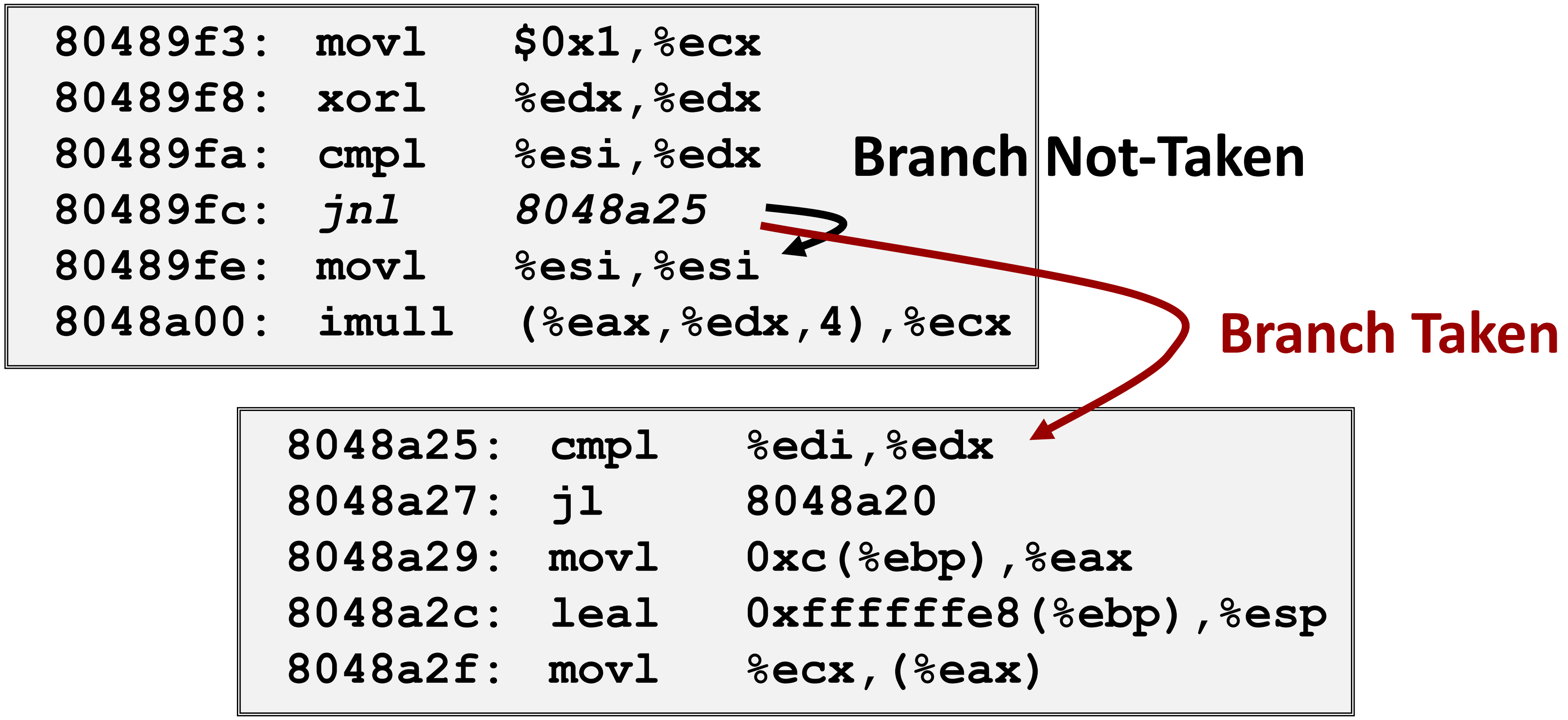
Executing

How to continue?

  - When encounters conditional branch, cannot reliably determine where to continue fetching

Branch Outcomes

- **When encounter conditional branch, cannot determine where to continue fetching**
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- **Cannot resolve until outcome determined by branch/integer unit**

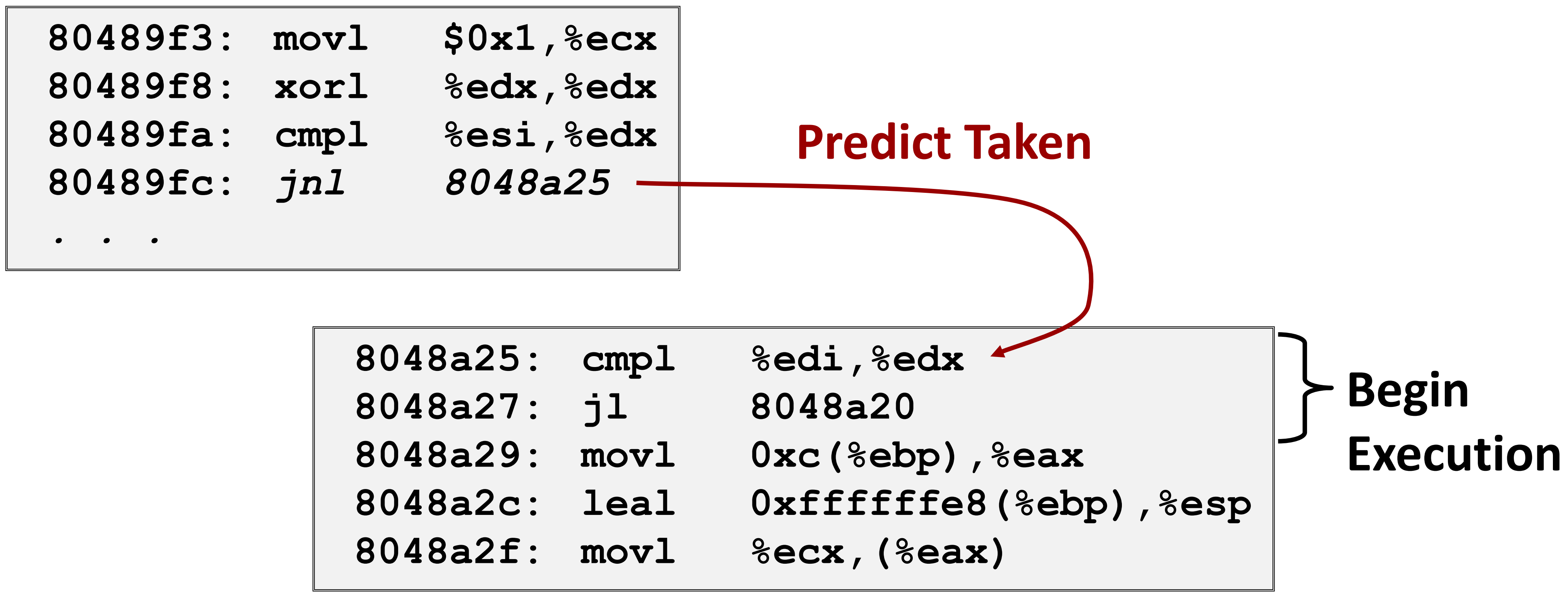


```
80489f3:  movl   $0x1,%ecx
80489f8:  xorl   %edx,%edx
80489fa:  cmpl   %esi,%edx
80489fc:  jnl    8048a25
80489fe:  movl   %esi,%esi
8048a00:  imull  (%eax,%edx,4),%ecx
```

```
8048a25:  cmpl   %edi,%edx
8048a27:  jl     8048a20
8048a29:  movl   0xc(%ebp),%eax
8048a2c:  leal   0xffffffe8(%ebp),%esp
8048a2f:  movl   %ecx,(%eax)
```

Branch Prediction

- **Idea**
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - But don’t actually modify register or memory data



```
 80489f3:  movl    $0x1,%ecx
 80489f8:  xorl    %edx,%edx
 80489fa:  cmpl    %esi,%edx
 80489fc:  jnl     8048a25
 . . .
```

```
 8048a25:  cmpl    %edi,%edx
 8048a27:  jl      8048a20
 8048a29:  movl    0xc(%ebp),%eax
 8048a2c:  leal    0xffffffe8(%ebp),%esp
 8048a2f:  movl    %ecx,(%eax)
```

Branch Prediction Through Loop

```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx
80488b9:    jl      80488b1
```

i = 98

```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx
80488b9:    jl      80488b1
```

i = 99

```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx
80488b9:    jl      80488b1
```

i = 100

```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx
80488b9:    jl      80488b1
```

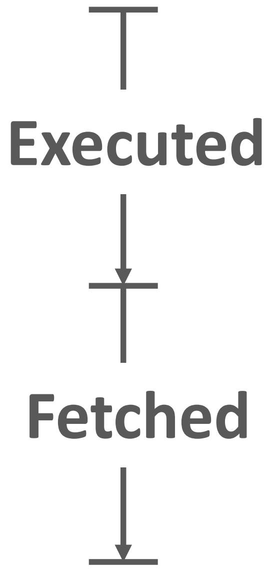
i = 101

Assume
vector length = 100

Predict Taken (OK)

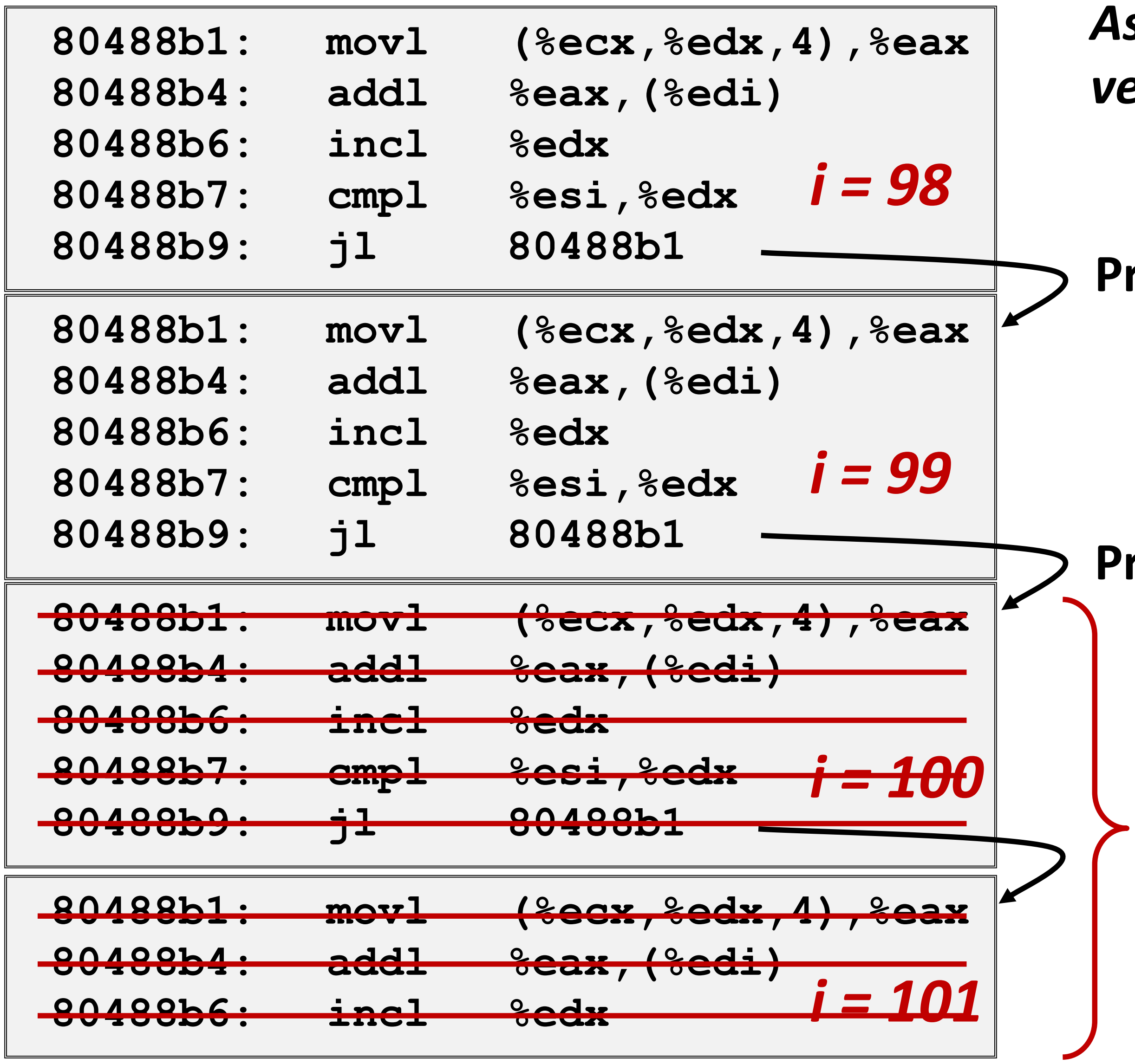
Predict Taken (Oops)

Read invalid location



Branch Misprediction Invalidation

Assume
vector length = 100



```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx
80488b9:    jl      80488b1
```

i = 98

Predict Taken (OK)

```
80488b1:    movl    (%ecx,%edx,4),%eax
80488b4:    addl    %eax,(%edi)
80488b6:    incl    %edx
80488b7:    cmpl    %esi,%edx
80488b9:    jl      80488b1
```

i = 99

Predict Taken (Oops)

~~80488b1: movl (%ecx,%edx,4),%eax~~
~~80488b4: addl %eax,(%edi)~~
~~80488b6: incl %edx~~
~~80488b7: cmpl %esi,%edx~~
~~80488b9: jl 80488b1~~

~~*i = 100*~~

~~80488b1: movl (%ecx,%edx,4),%eax~~
~~80488b4: addl %eax,(%edi)~~
~~80488b6: incl %edx~~

~~*i = 101*~~

Invalidate

Branch Misprediction Recovery

```
80488b1:   movl    (%ecx,%edx,4),%eax
80488b4:   addl    %eax,(%edi)
80488b6:   incl    %edx
80488b7:   cmpl    %esi,%edx
80488b9:   jl      80488b1
80488bb:   leal    0xffffffe8(%ebp),%esp
80488be:   popl    %ebx
80488bf:   popl    %esi
80488c0:   popl    %edi
```

i = 99

Definitely not taken

- **Performance Cost**
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter