CS33 Discussion

Week 5

How does a processor execute an instruction?

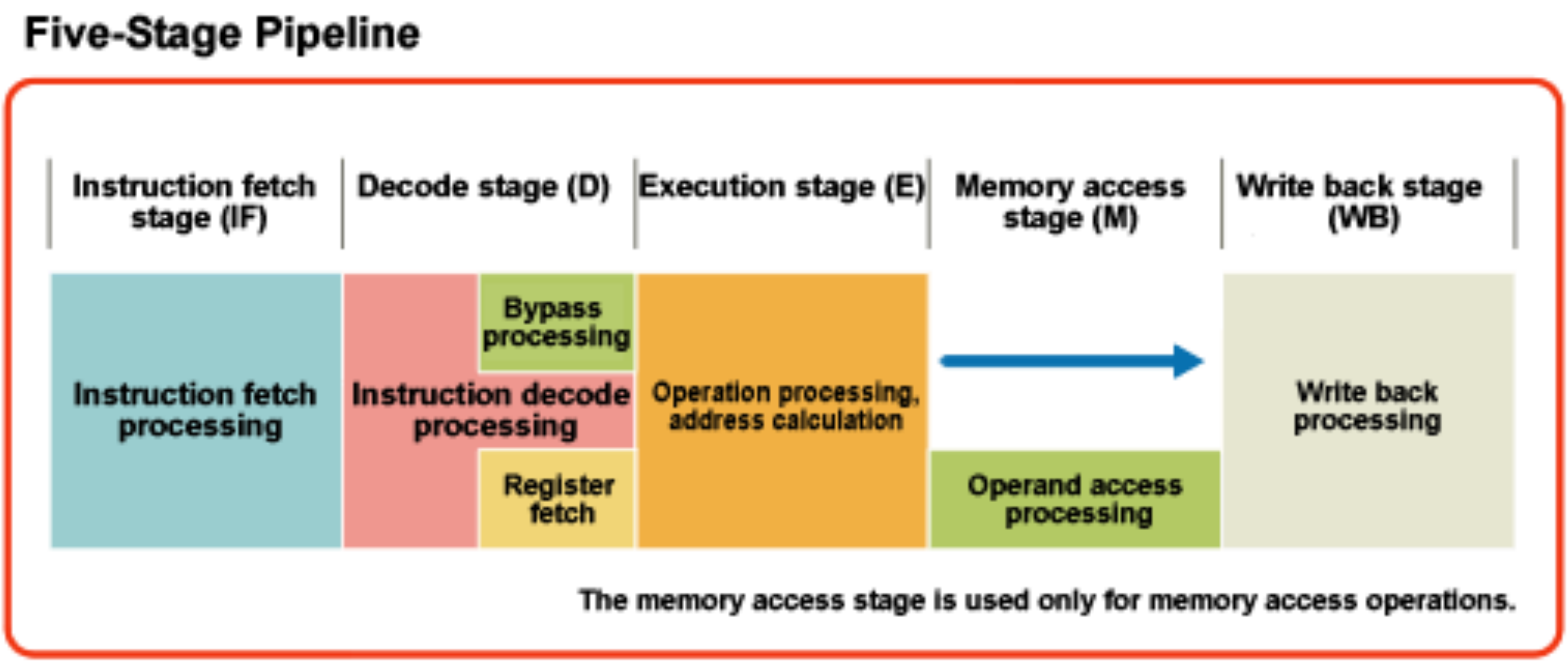
Instruction Execution

- Step by step
  - Where is the instruction?
    - Memory -> CPU: Instruction Fetch
  - What is the instruction?
    - String of 0's and 1's -> Meaningful operations when decoded
  - Where are the source operands coming from?
    - Read from register file/memory
  - Do something
    - Arithmetic/Logic operations, Effective address calculation, Branch condition evaluation, …
  - Where is the result going?
    - Write to register file/memory

Typical Instruction Execution Flow

- Step 1: Instruction Fetch (IF)
  - Fetch the instruction from memory to CPU
- Step 2: Instruction Decode (ID)
  - Decode AND read source operands from register file
- Step 3: Execution (EX)
  - Do the arithmetic/logic operations of the instruction
- Step 4: Memory Access (MEM)
  - Load/Store data from/to memory
- Step 5: Write Back
  - Write the destination operand back to register file

Typical Instruction Execution Flow



Instruction Execution: Example 1

- Add %eax, %ebx
  - IF: IR <= Fetch[PC], PC <= Next PC
  - ID: Decode[IR], A <= Value[eax], B <= Value[ebx]
  - EX: SUM <= A+B
  - MEM: Do nothing
  - WB: Value[ebx] <= SUM

Instruction Execution: Example 2

- Mov 12(%eax), %ebx
  - IF: IR <= Fetch[PC], PC <= Next PC
  - ID: Decode[IR], A <= Value[eax], B <= 12
  - EX: Address <= A+B
  - MEM: Data <= Load[Address]
  - WB: Value[ebx] <= Data

Instruction Execution: Example 3

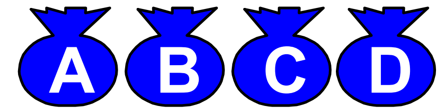
- JE Label
  - IF: IR <= Fetch[PC], PC <= Next PC
  - ID: Decode[IR], A <= Value[ZeroFlag]
  - EX: if (A==1) PC <= Label
  - MEM: Do nothing
  - WB: Do nothing

Execution Time

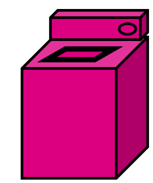
- Latency
  - Given the 5-step instruction execution flow, if each step costs 1 second, how long does it take to execute one instruction?
- Throughput
  - Given the 5-step instruction execution flow, how many instructions can be executed within one hour if a CPU executes them one by one?

Gotta Do Laundry

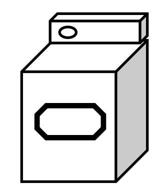
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away



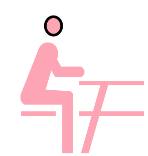
  - Washer takes 30 minutes



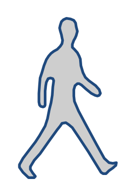
  - Dryer takes 30 minutes



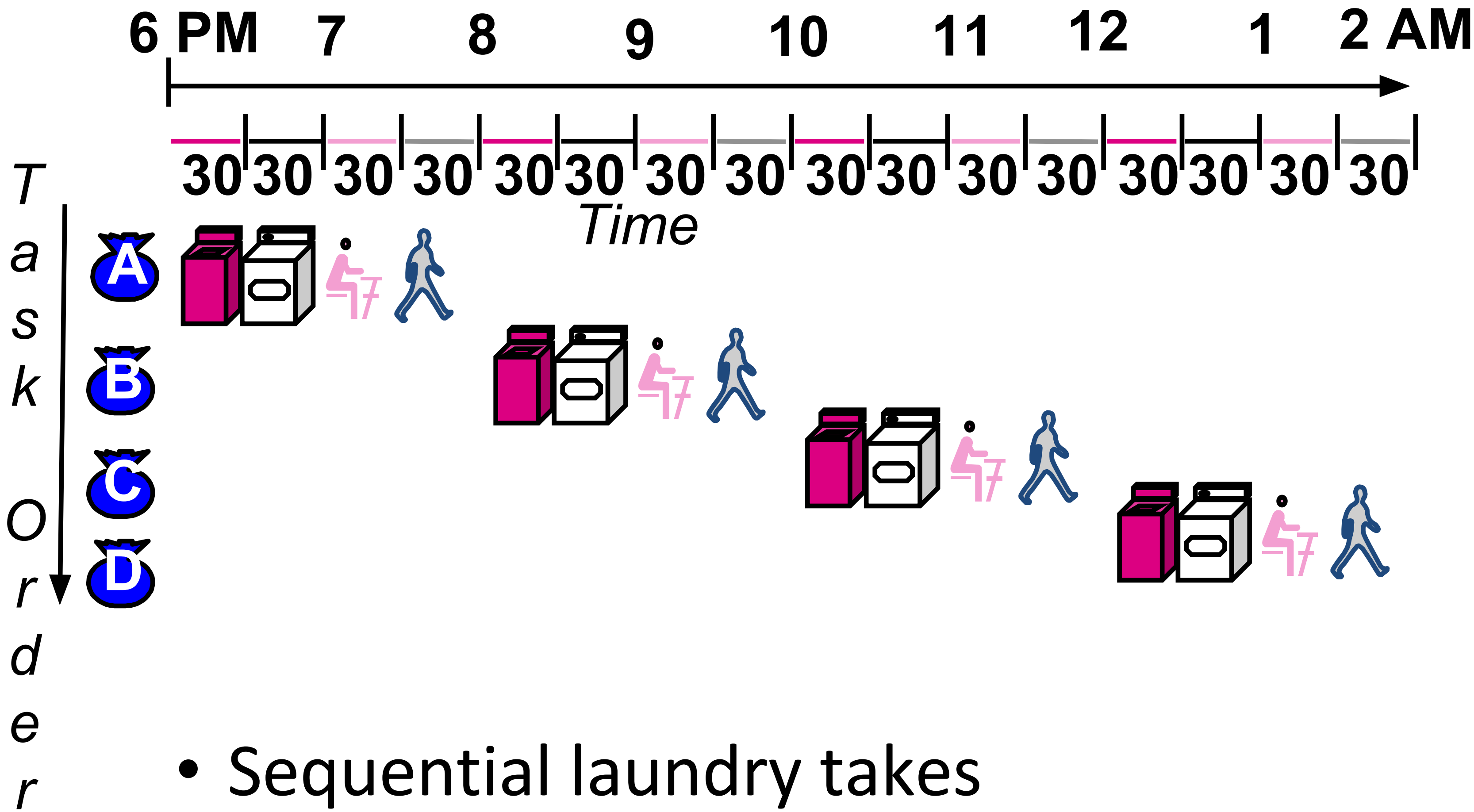
  - “Folder” takes 30 minutes



  - “Stasher” takes 30 minutes to put clothes into drawers

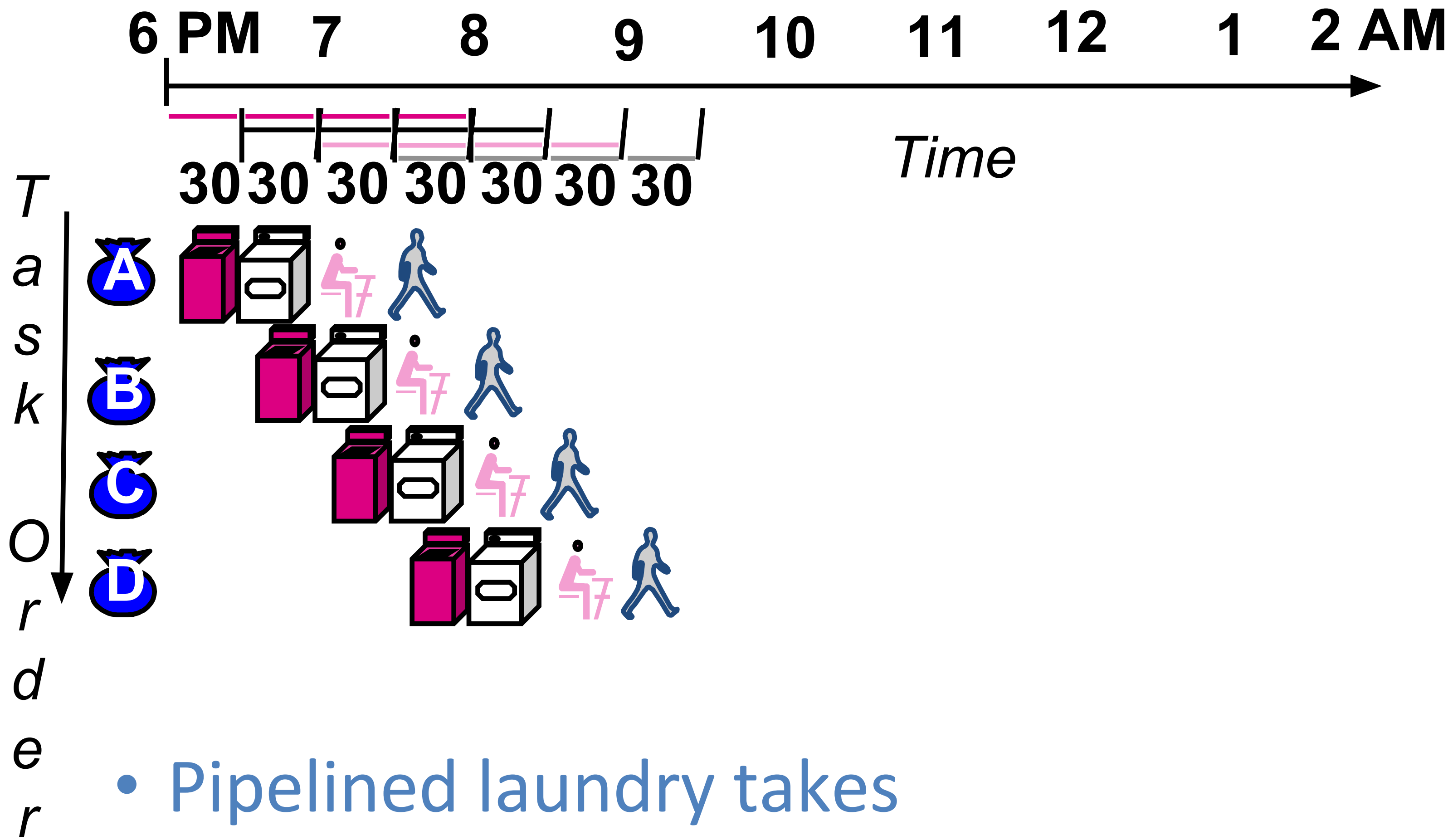


Sequential Laundry



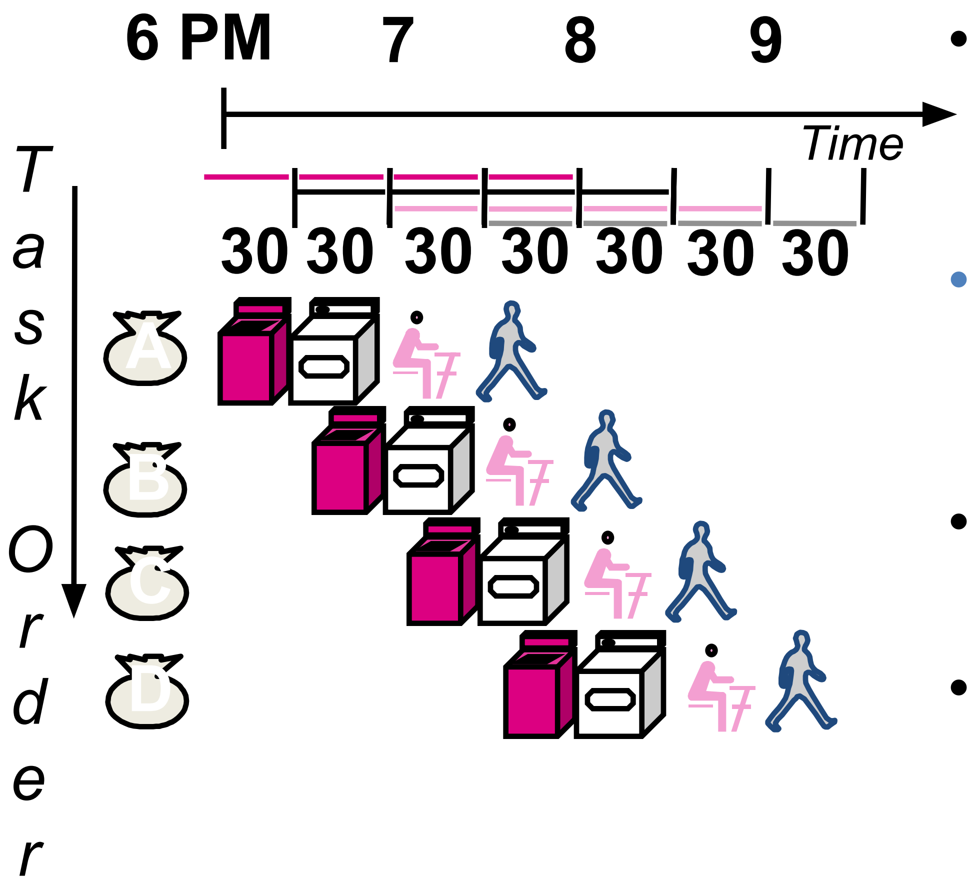
- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry



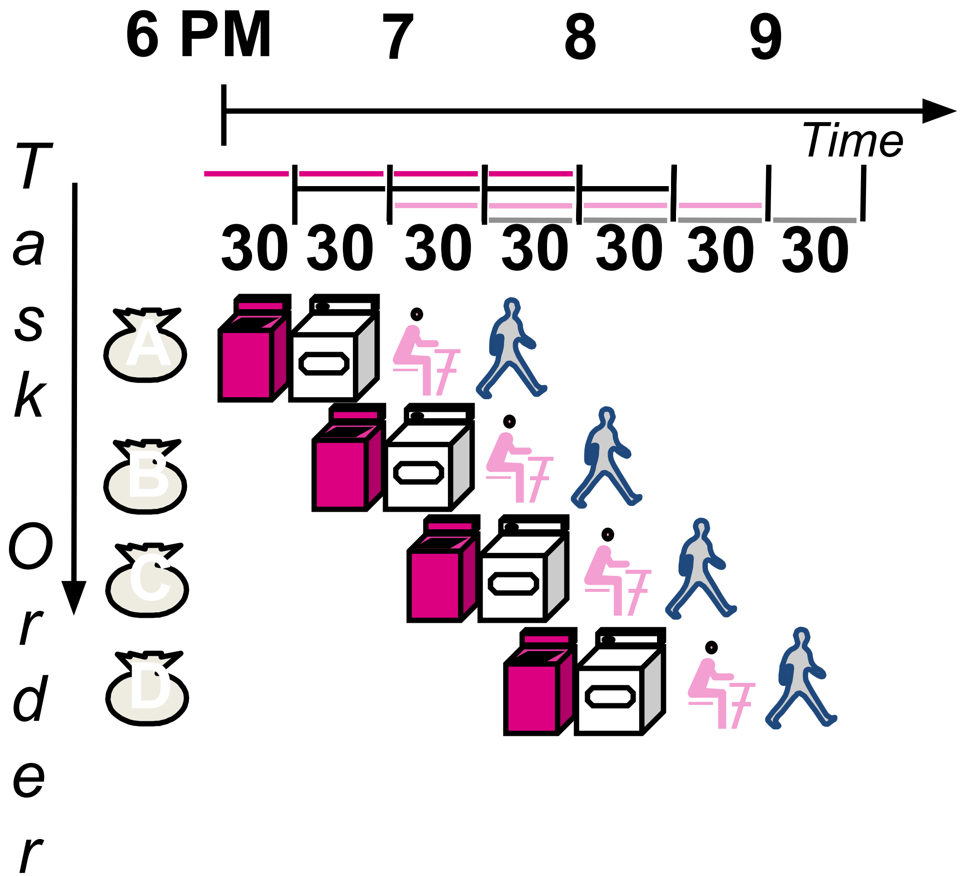
- Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons (1/2)



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Time to "fill" pipeline and time to "drain" it reduces speedup: 2.3X v. 4X in this example

Pipelining Lessons (2/2)



- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup

Execution Time

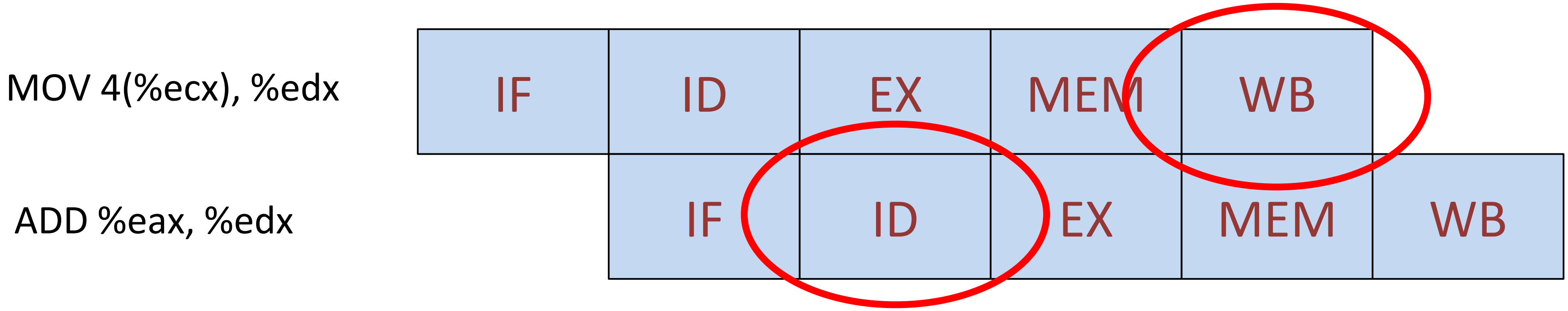
- Latency
  - Given the 5-step instruction execution flow, if each step costs 1 second, how long does it take to execute one instruction?
- Throughput
  - Given the 5-step instruction execution flow, how many instructions can be executed within one hour if a CPU executes them PIPELINEDLY?

Can a pipeline always work perfectly?

Pipeline Hazard

```
Label:
    Mov 4(%ecx), %edx
    Add %eax, %edx
    Cmp %ebx, %edx
    Je Label
    ….
```

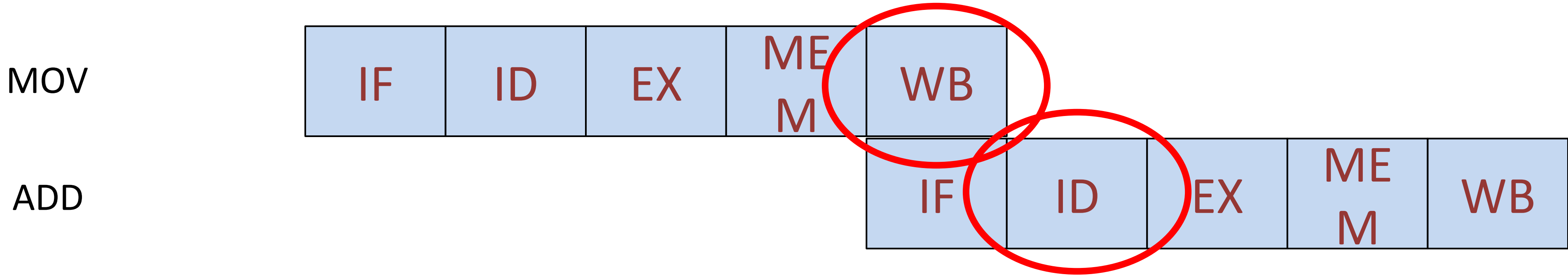
Pipeline Hazard



- Mov 4(%ecx), %edx
  - WB: %edx <= Data (fifth second)
- Add %eax, %edx
  - ID: A<= %edx (third second)

Can Add get the correct operands?

Pipeline Hazard

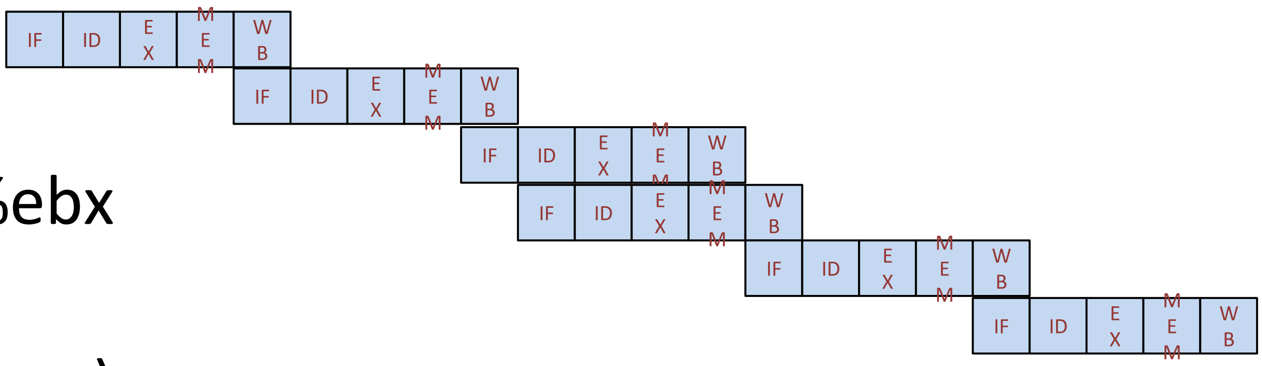


How many seconds got lost?

Loop Unrolling

For (i=0; i<100; i++) {
    A[i] += 7;
}

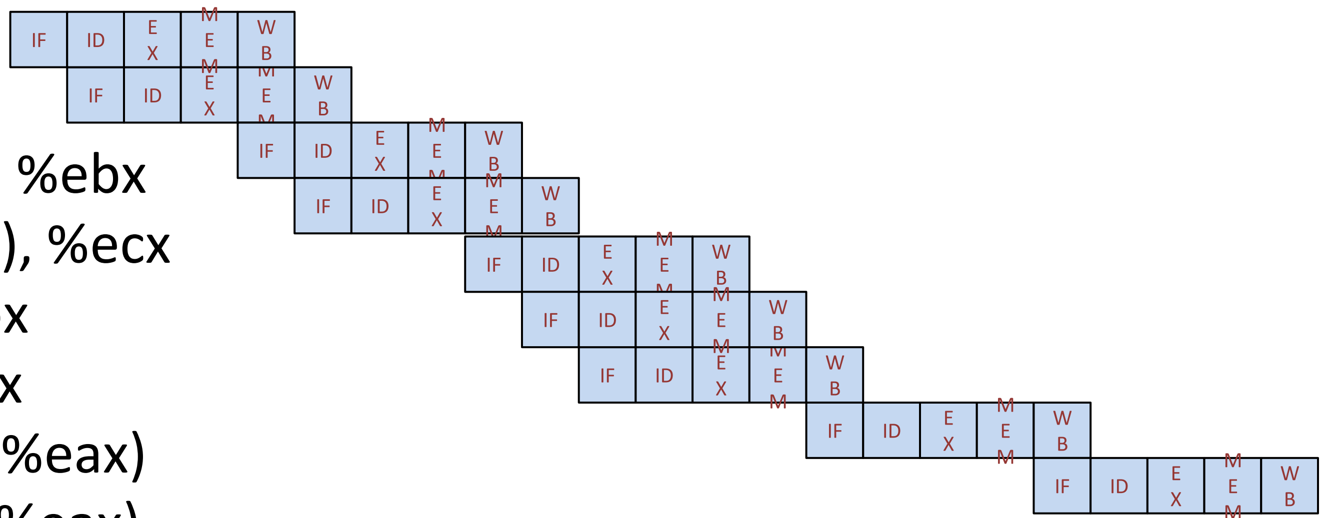
```
Loop:
    mov (%eax), %ebx
    add $7, %ebx
    mov %ebx, (%eax)
    add $4, %eax
    cmp $400, %eax
    jne Loop
    ...
```



Cycle per Element?

Loop Unrolling

```
For (i=0; i<100; i+=2) {
    A[i] += 7;
    A[i+1] += 7;
}
Loop:
    mov (%eax), %ebx
    mov 4(%eax), %ecx
    add $7, %ebx
    add $7, %ecx
    mov %ebx, (%eax)
    mov %ecx, (%eax)
    add $8, %eax
    cmp $400, %eax
    jne Loop
    …
```

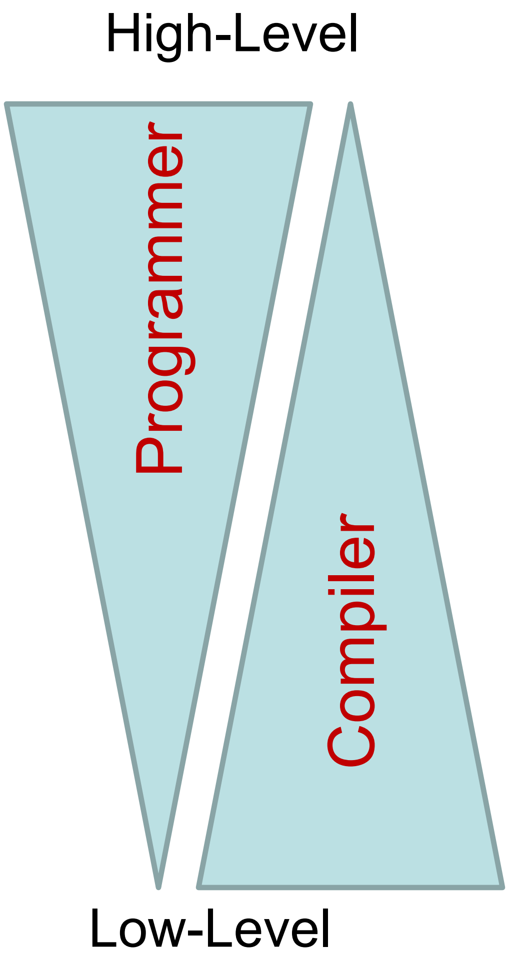


Cycle per Element?

Code Optimization

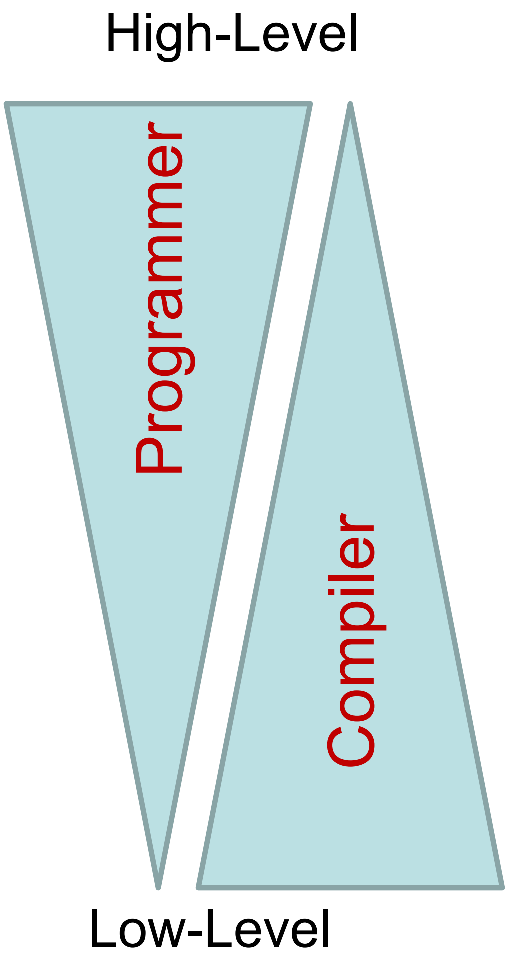
Roles of Programmer vs Compiler

- Programmer:
  - Choice of algorithm
    - Quick Sort? Insert Sort?
  - Manual application of some optimizations
  - Choice of program structure that's amenable to optimization
  - Avoidance of "optimization blockers"

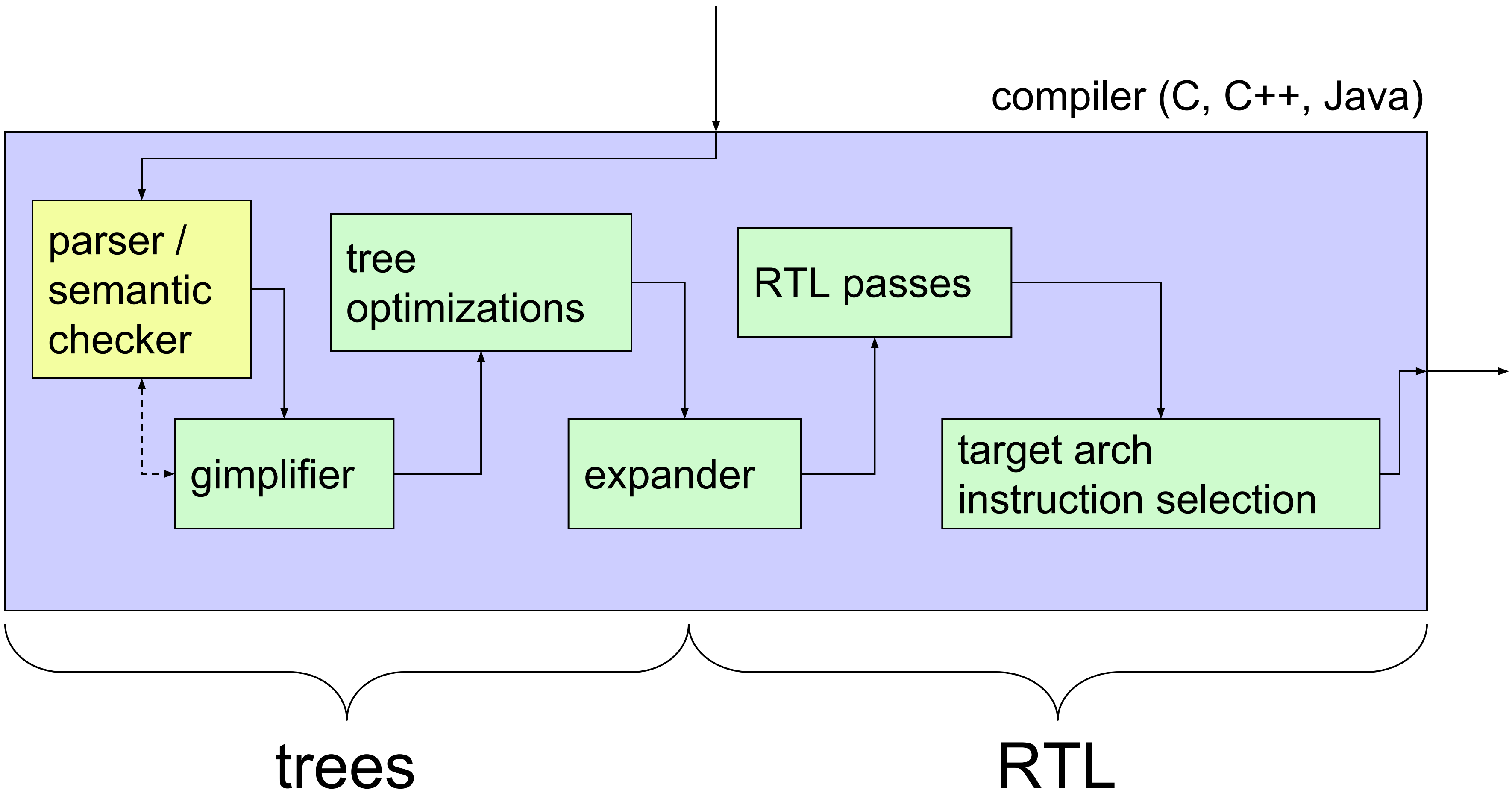


Roles of Programmer vs Compiler

- Optimizing Compiler
  - Applies transformations that preserve semantics, but reduce amount of, or time spent in computations
  - Provides efficient mapping of code to machine:
    - Selects and orders code
    - Performs register allocation
  - Usually consists of multiple stages



Inside GCC



Source: Deters & Cytron, OOPSLA 2006

Limitations of Optimizing Compilers

- Fundamentally, must emit code that implements specified semantics under *all* conditions
  - Can't apply optimizations even if they would only change behavior in corner case a programmer may not think of
  - Due to memory aliasing
  - Due to unseen procedure side-effects
- Do not see beyond current compilation unit
- Intraprocedural analysis typically more extensive (since cheaper) than interprocedural analysis
- Usually base decisions on static information

Optimizations

- Copy Propagation
- Code Motion
- Strength Reduction
- Common Subexpression Elimination
- Eliminating Memory Accesses
  - Through use of registers
- Inlining

Getting the compiler to optimize

- **-O0** ("O zero")
  - This is the default: Do not optimize
- **-O1**
  - Apply optimizations that can be done quickly
- **-O2**
  - Apply more expensive optimizations. That's a reasonable default for running production code. Typical ratio between –O2 and –O0 is 5-20.
- **-O3**
  - Apply even more expensive optimizations
- **-Os**
  - Optimize for code size
- See 'info gcc' for list which optimizations are enabled when; note that –f switches may enable additional optimizations that are not included in –O
- Note: ability to debug code symbolically under gdb decreases with optimization level; usually use –O0 –g or –O1 -g
- NB: other compilers use different switches – some have up to –O7

Code Motion

- Do not repeat computations if result is known
- Usually out of loops (“code hoisting”)

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

Strength Reduction

- Substitute lower cost operation for more expensive one
  - E.g., replace 48*x with (x << 6) – (x << 4)
  - Often machine dependent

Common Subexpression Elimination

3 multiplications: i*n, (i–1)*n, (i+1)*n

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j];
down =  val[(i+1)*n + j];
left =  val[i*n   + j-1];
right = val[i*n   + j+1];
sum = up + down + left + right;
```

1 multiplication: i*n

```
int inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

- Reuse already computed expressions

Inlining

- Substitute body of called function into the caller
  - *before subsequent optimizations are applied*
- Two ways for programmers to inline functions
  - #define MAX_INT((a), (b)) ((a)>(b)?(a):(b))
  - Inline int max(int a, int b) {return a>b?a:b;}
- Current compilers do this aggressively
- Almost never a need for doing this manually

Inlining Example

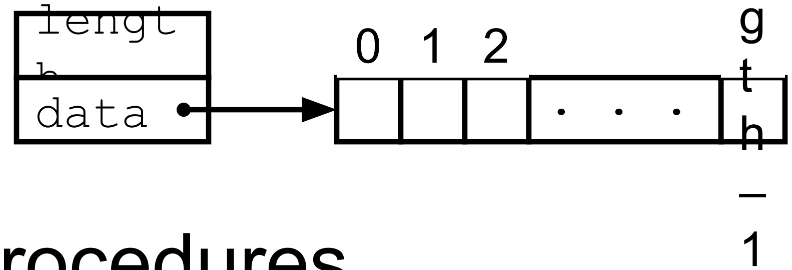
```
void sp1(double *x, double *y,
         double *sum, double *prod)
{
    *sum =  *x + *y;
    *prod = *x * *y;
}

double outersp1(double *x,
                double *y)
{
    double sum, prod;

    sp1(x, y, &sum, &prod);
    return sum > prod ? sum : prod;
}
```

```
outersp1:
    movsd   (%rdi), %xmm1
    movsd   (%rsi), %xmm2
    movapd  %xmm1, %xmm0
    mulsd   %xmm2, %xmm1
    addsd   %xmm2, %xmm0
    maxsd   %xmm1, %xmm0
    ret
```

Case Study: Vector ADT



- Procedures

  `vec_ptr new_vec(int len)`
    - Create vector of specified length

  `int get_vec_element(vec_ptr v, int index, int *dest)`
    - Retrieve vector element, store at *dest
    - Return 0 if out of bounds, 1 if successful

  `int *get_vec_start(vec_ptr v)`
    - Return pointer to start of vector data

  – Similar to array implementations in Pascal, ML, Java
    - E.g., always do bounds checking

Optimization Example

```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

- Procedure
  - Compute sum of all elements of vector
  - Store result at destination location

Optimization Example

```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

- Procedure
  - Compute sum of all elements of integer vector
  - Store result at destination location
  - Vector data structure and operations defined via abstract data type
- Pentium II/III Performance: Clock Cycles / Element
  - 42.06 (Compiled -O0) 31.25 (Compiled -O2)

Understanding Loop

```
void combine1-goto(vec_ptr v, int *dest)
{
    int i = 0;
    int val;
    *dest = 0;
    if (i >= vec_length(v))
      goto done;
  loop:
    get_vec_element(v, i, &val);
    *dest += val;
    i++;
    if (i < vec_length(v))
      goto loop
  done:
}
```

1 iteration

- Inefficiency
  - Procedure vec_length called every iteration
  - Even though result always the same

Move `vec_length` Call Out of Loop

```
void combine2(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

- Optimization
  - Move call to `vec_length` out of inner loop
    - Value does not change from one iteration to next
    - Code motion
  - CPE: 20.66 (Compiled -O2)
    - `vec_length` requires only constant time, but significant overhead

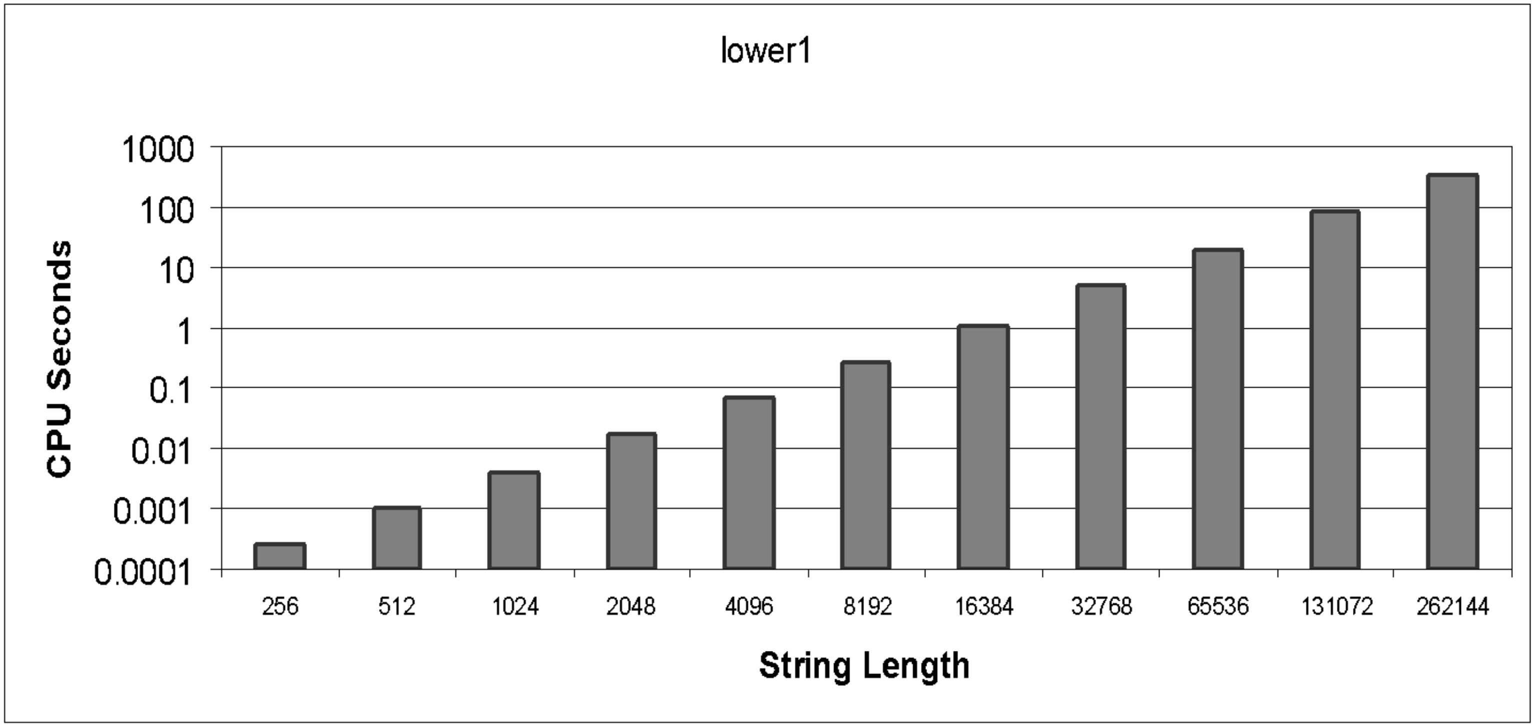
Code Motion Example #2

```
void lower(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

- Convert string from upper to lower
- Here: asymptotic complexity becomes O (n^2)!

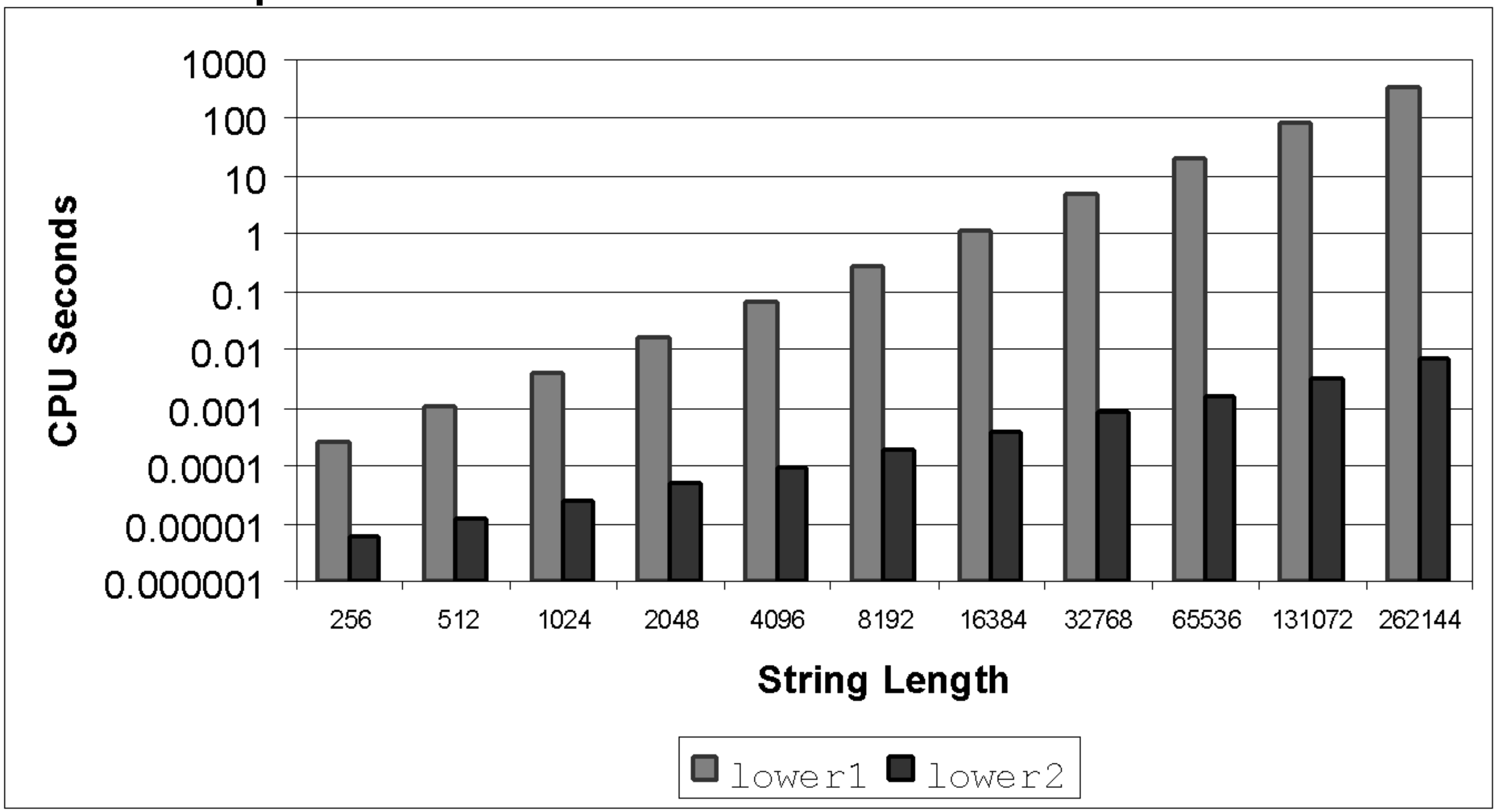
Lower Case Conversion Performance

– Time quadruples when double string length
– Quadratic performance



Performance after Code Motion

– Time doubles when double string length
– Linear performance



Optimization Blocker: Procedure Calls

- *Why couldn't the compiler move* `vec_len` *or* `strlen` *out of the inner loop?*
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure `lower` could interact with `strlen`
- *What if compiler looks at code? Or inlines them?*
  - even then, compiler may not be able to prove that the same result is obtained, or the possibility of aliasing may require repeating the operation; and compiler must preserve any side-effects
  - interprocedural optimization is expensive, but compilers are continuously getting better at it
    - For instance, take into account if a function reads or writes to global memory
  - Today's compilers are different from the compilers 5 years ago and will be different from those 5 years from now

Remove Bounds Checking

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
}
```

- Optimization
  - Avoid procedure call to retrieve each vector element
    - Get pointer to start of array before loop
    - Within loop just do pointer reference
    - Not as clean in terms of data abstraction
  - CPE: 6.00 (Compiled -O2)
    - Procedure calls are expensive!
    - Bounds checking is expensive

Eliminate Unneeded Memory Refs

```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

- Optimization
  - Don't need to store in destination until end
  - Local variable `sum` held in register
  - Avoids 1 memory read, 1 memory write per cycle
  - CPE: 2.00 (Compiled -O2)
    - Memory references are expensive!

Detecting Unneeded Memory Refs.

Combine3

```
.L18:
   movl (%ecx,%edx,4),%eax
   addl %eax,(%edi)
   incl %edx
   cmpl %esi,%edx
   jl .L18
```

Combine4

```
.L24:
   addl (%eax,%edx,4),%ecx

   incl %edx
   cmpl %esi,%edx
   jl .L24
```

- Performance
  - Combine3
    - 5 instructions in 6 clock cycles
    - `addl` must read and write memory
  - Combine4
    - 4 instructions in 2 clock cycles

Pointer Code

```
void combine4p(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int *dend = data+length;
  int sum = 0;
  while (data < dend) {
    sum += *data;
    data++;
  }
  *dest = sum;
}
```

Big question:
Should you rewrite your array code as pointer code to “help” the compiler?

- Optimization
  - Use pointers rather than array references
  - CPE: 3.00 (Compiled -O2)
    - Oops! Worse than the best array version

Warning: Some compilers do better job optimizing array code

Pointer vs. Array Code

- Difficult to predict which would be faster
- Compiler may transform array to pointer form if it deems it useful
- Compiler as a rule optimizes array code as good or better as it does pointer code
- Writing as array code allows use of index variable in index-based address modes
- Should prefer array form for readability


thank you!