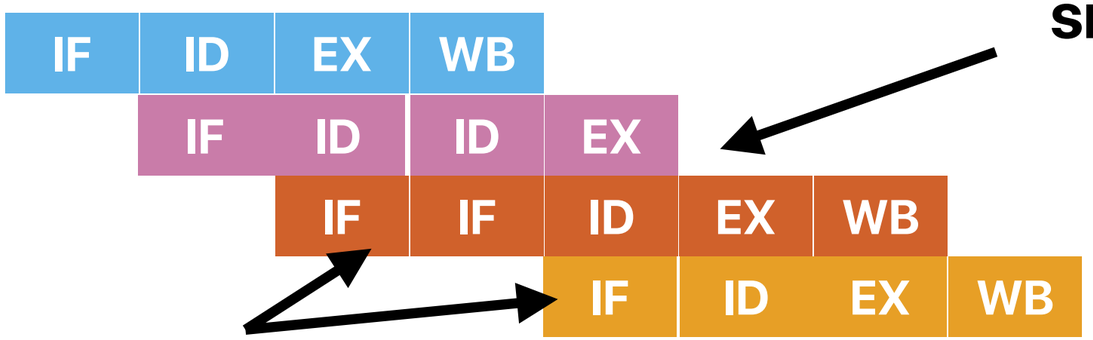
Modern Processor Design (IV): Can't Hardly Wait

Hung-Wei Tseng

Recap: Branch Prediction

```
① cmpq %rdx, %rdi
② jne  .L3
③ ret
④ something …
```

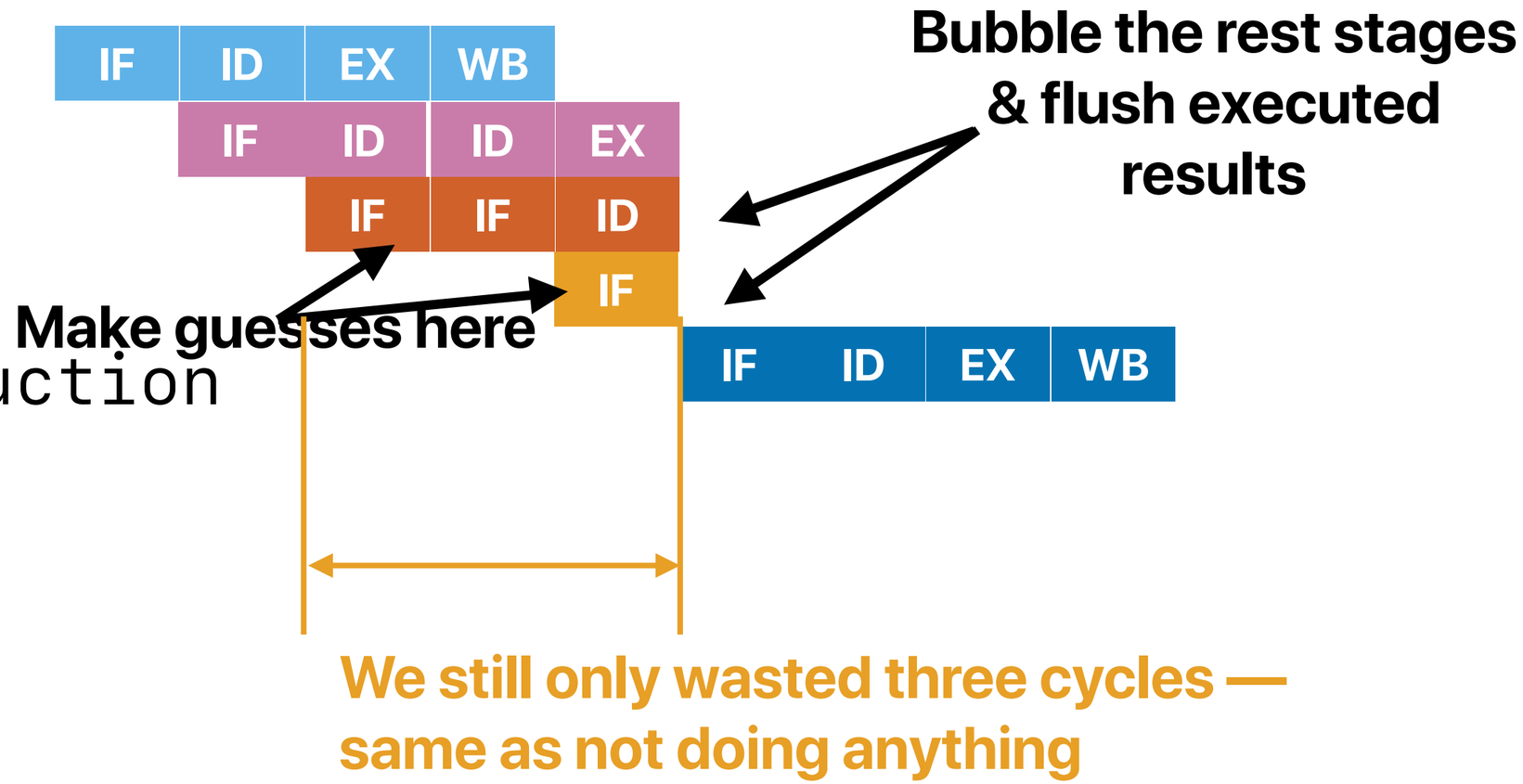


We can execute smoothly if we guess right!

Make guesses here

Recap: What if we are wrong?

```
① cmpq %rdx, %rdi
② jne  .L3
③ ret
④ something …
⑤ the right instruction
```



Recap: Branch predictors in processors

- The Intel Pentium MMX, Pentium II, and Pentium III have local branch predictors with a local 4-bit history and a local pattern history table with 16 entries for each conditional jump.
- Global branch prediction is used in Intel Pentium M, Core, Core 2, and Silvermont-based Atom processors.
- Tournament predictor is used in DEC Alpha, AMD Athlon processors
- The AMD Ryzen multi-core processor's Infinity Fabric and the Samsung Exynos processor include a perceptron based neural branch predictor.

Demo: Popcount

- Given a 64-bit integer number, find the number of 1s in its binary representation.
- Example 1:
  Input: 59487
  Output: 10
  Explanation: 59487's binary representation is 0b1110100001011111

```
int main(int argc, char *argv[]) {

    uint64_t key = 0xdeadbeef;

    int count = 1000000000;
    uint64_t sum = 0;

    for (int i=0; i < count; i++)
    {
        sum += popcount(RandLFSR(key));
    }
    printf("Result: %lu\n", sum);
    return sum;
}
```

Five implementations

- Which of the following implementations will perform the best on modern pipeline processors?

A

```
inline int popcount(uint64_t x){
  int c=0;
  while(x)  {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

B

```
inline int popcount(uint64_t x) {
   int c = 0;
   while(x)     {
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
   }
   return c;
}
```

C

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     while(x)     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
}
```

D

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
  }
```

E

```
inline int popcount(uint64_t x) {
     int c = 0;
     for (uint64_t i = 0; i < 16; i++)
     {
         switch((x & 0xF))
         {
             case 1: c+=1; break;
             case 2: c+=1; break;
             case 3: c+=2; break;
             case 4: c+=1; break;
             case 5: c+=2; break;
             case 6: c+=2; break;
             case 7: c+=3; break;
             case 8: c+=1; break;
             case 9: c+=2; break;
             case 10: c+=2; break;
             case 11: c+=3; break;
             case 12: c+=2; break;
             case 13: c+=3; break;
             case 14: c+=3; break;
             case 15: c+=4; break;
             default: break;
         }
         x = x >> 4;
     }
     return c;
}
```

Why is B better than A?

A

```
inline int popcount(uint64_t x){
  int c=0;
  while(x)  {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

```
movl    %eax, %ecx
andl    $1, %ecx
addl    %ecx, %edx
shrq    %rax
jne     .L6
```

5*n instructions

B

```
inline int popcount(uint64_t x) {
   int c = 0;
   while(x)     {
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
     c += x & 1;
     x = x >> 1;
   }
   return c;
}
```

15*(n/4) = 3.75*n instructions

```
movl    %ecx, %eax
andl    $1, %eax
addl    %edx, %eax
movq    %rcx, %rdx
shrq    %rdx
andl    $1, %edx
addl    %eax, %edx
movq    %rcx, %rax
shrq    $2, %rax
andl    $1, %eax
addl    %edx, %eax
movq    %rcx, %rdx
shrq    $3, %rdx
andl    $1, %edx
addl    %eax, %edx
shrq    $4, %rcx
jne     .L6
```

Only one branch for four iterations in A

Why is C better than B?

- How many of the following statements explains the reason why B outperforms C with compiler optimizations
  - ① C has lower dynamic instruction count than B — C only needs one load, one add, one shift, the same amount of iterations
  - ② C has significantly lower branch mis-prediction rate than B — the same number being predicted.
  - ③ C has significantly fewer branch instructions than B — the same amount of branches
  - ④ C can incur fewer data hazards — Probably not. In fact, the load may have negative effect without architectural supports

A. 0

B. 1

C. 2

D. 3

E. 4

C

```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    while(x)     {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

B

```
inline int popcount(uint64_t x) {
    int c = 0;
    while(x)     {
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
      c += x & 1;
      x = x >> 1;
    }
    return c;
}
```

Outline

- Programming on processors with advanced branch predictors (cont.)
- Data hazards
- Hardware optimizations for data hazards

https://www.pollev.com/hungweitseng close in 1:30

Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① D has lower dynamic instruction count than C
  - ② D has significantly lower branch mis-prediction rate than C
  - ③ D has significantly fewer branch instructions than C
  - ④ D can incur fewer data hazards than C

A. 0
B. 1
C. 2
D. 3
E. 4

C

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     while(x)     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
}
```

D

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
 }
```

Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① D has lower dynamic instruction count than C
  - ② D has significantly lower branch mis-prediction rate than C
  - ③ D has significantly fewer branch instructions than C
  - ④ D can incur fewer data hazards than C

A. 0
B. 1
C. 2
D. 3
E. 4

C

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     while(x)     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
}
```

D

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
 }
```

Loop unrolling eliminates all branches!

D

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
 }
```

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
          c += table[(x & 0xF)];
          x = x >> 4;
     return c;
}
```

Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① D has lower dynamic instruction count than C — Compiler can do loop unrolling — no branches
  - ② D has significantly lower branch mis-prediction rate than C — Could be
  - ③ D has significantly fewer branch instructions than C — maybe eliminated through loop unrolling...
  - ④ D can incur fewer data hazards than C — about the same

A. 0
B. 1
C. 2
D. 3
E. 4

C

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     while(x)     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
}
```

D

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
 }
```

https://www.pollev.com/hungweitseng close in 1:30

Why is E the slowest?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① E has the most dynamic instruction count
  - ② E has the highest branch mis-prediction rate
  - ③ E has the most branch instructions
  - ④ E can incur the most data hazards than others

A. 0
B. 1
C. 2
D. 3
E. 4

D

```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    for (uint64_t i = 0; i < 16; i++)
    {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

E

```
inline int popcount(uint64_t x) {
    int c = 0;
    for (uint64_t i = 0; i < 16; i++)
    {
        switch((x & 0xF))
        {
            case 1: c+=1; break;
            case 2: c+=1; break;
            case 3: c+=2; break;
            case 4: c+=1; break;
            case 5: c+=2; break;
            case 6: c+=2; break;
            case 7: c+=3; break;
            case 8: c+=1; break;
            case 9: c+=2; break;
            case 10: c+=2; break;
            case 11: c+=3; break;
            case 12: c+=2; break;
            case 13: c+=3; break;
            case 14: c+=3; break;
            case 15: c+=4; break;
            default: break;
        }
        x = x >> 4;
    }
    return c;
}
```

Why is E the slowest?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① E has the most dynamic instruction count
  - ② E has the highest branch mis-prediction rate
  - ③ E has the most branch instructions
  - ④ E can incur the most data hazards than others

A. 0
B. 1
C. 2
D. 3
E. 4

D

```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    for (uint64_t i = 0; i < 16; i++)
    {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

E

```
inline int popcount(uint64_t x) {
    int c = 0;
    for (uint64_t i = 0; i < 16; i++)
    {
        switch((x & 0xF))
        {
            case 1: c+=1; break;
            case 2: c+=1; break;
            case 3: c+=2; break;
            case 4: c+=1; break;
            case 5: c+=2; break;
            case 6: c+=2; break;
            case 7: c+=3; break;
            case 8: c+=1; break;
            case 9: c+=2; break;
            case 10: c+=2; break;
            case 11: c+=3; break;
            case 12: c+=2; break;
            case 13: c+=3; break;
            case 14: c+=3; break;
            case 15: c+=4; break;
            default: break;
        }
        x = x >> 4;
    }
    return c;
}
```

Why is E the slowest?

E

```
inline int popcount(uint64_t x) {
    int c = 0;
    for (uint64_t i = 0; i < 16; i++)
    {
        switch((x & 0xF))
        {
            case 1: c+=1; break;
            case 2: c+=1; break;
            case 3: c+=2; break;
            case 4: c+=1; break;
            case 5: c+=2; break;
            case 6: c+=2; break;
            case 7: c+=3; break;
            case 8: c+=1; break;
            case 9: c+=2; break;
            case 10: c+=2; break;
            case 11: c+=3; break;
            case 12: c+=2; break;
            case 13: c+=3; break;
            case 14: c+=3; break;
            case 15: c+=4; break;
            default: break;
        }
        x = x >> 4;
    }
    return c;
}
```

```
.L11:
        movq    %r9, %rcx
        andl    $15, %ecx
        movslq  (%r8,%rcx,4), %rcx
        addq    %r8, %rcx
        notrack jmp     *%rcx
.L7:
        .long   .L5-.L7
        .long   .L10-.L7
        .long   .L10-.L7
        .long   .L9-.L7
        .long   .L10-.L7
        .long   .L9-.L7
        .long   .L9-.L7
        .long   .L8-.L7
        .long   .L10-.L7
        .long   .L9-.L7
        .long   .L9-.L7
        .long   .L8-.L7
        .long   .L9-.L7
        .long   .L8-.L7
        .long   .L8-.L7
        .long   .L6-.L7
.L8:
        addl    $3, %eax
.L5:
        shrq    $4, %r9
        subq    $1, %rsi
        jne     .L11
        cltq
        addq    %rax, %rbx
        subl    $1, %edi
        jne     .L12
```

```
.L9:
        .cfi_restore_state
        addl    $2, %eax
        jmp     .L5
        .p2align 4,,10
        .p2align 3
.L10:
        addl    $1, %eax
        jmp     .L5
        .p2align 4,,10
        .p2align 3
.L6:
        addl    $4, %eax
        jmp     .L5
```

Why is E the slowest?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
  - ① E has the most dynamic instruction count
  - ② E has the highest branch mis-prediction rate
  - ③ E has the most branch instructions
  - ④ E can incur the most data hazards than others

A. 0

B. 1

C. 2

D. 3

E. 4

D

```
inline int popcount(uint64_t x) {
     int c = 0;
     int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
     for (uint64_t i = 0; i < 16; i++)
     {
         c += table[(x & 0xF)];
         x = x >> 4;
     }
     return c;
 }
```

E

```
inline int popcount(uint64_t x) {
     int c = 0;
     for (uint64_t i = 0; i < 16; i++)
     {
         switch((x & 0xF))
         {
             case 1: c+=1; break;
             case 2: c+=1; break;
             case 3: c+=2; break;
             case 4: c+=1; break;
             case 5: c+=2; break;
             case 6: c+=2; break;
             case 7: c+=3; break;
             case 8: c+=1; break;
             case 9: c+=2; break;
             case 10: c+=2; break;
             case 11: c+=3; break;
             case 12: c+=2; break;
             case 13: c+=3; break;
             case 14: c+=3; break;
             case 15: c+=4; break;
             default: break;
         }
         x = x >> 4;
     }
     return c;
 }
```

Hardware acceleration

- Because popcount is important, both intel and AMD added a POPCNT instruction in their processors with SSE4.2 and SSE4a
- In C/C++, you may use the intrinsic "_mm_popcnt_u64" to get # of "1"s in an unsigned 64-bit number
  - You need to compile the program with -m64 -msse4.2 flags to enable these new features

```
#include <smmintrin.h>
inline int popcount(uint64_t x) {
     int c = _mm_popcnt_u64(x);
     return c;
}
```

Summary of popcounts

	ET	IC	IPC/ILP	# of branches	Branch mis-prediction rate
A	22.21	332 Trillions	2.88	65 Trillions	1.13%
B	12.29	287 Trillions	4.52	17 Trillions	0.04%
C	5.01	102 Trillions	3.95	17 Trillions	0.04%
D	3.73	80 Trillions	4.13	1 Trillions	~0%
E	54.4	173 Trillions	0.61	44 Trillions	18.6%
SSE4.2	1.57	22 Trillions	2.7	1 Trillions	~0%

Recap: Which swap is faster?

A

```
void regswap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

B

```
void xorswap(int* a, int* b) {
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}
```

- Both version A and B swaps content pointed by a and b correctly. Which version of code would have better performance?
  A. Version A
  B. Version B
  C. They are about the same (sometimes A is faster, sometimes B is)

Data hazards

Data hazards

- An instruction currently in the pipeline cannot receive the "logically" correct value for execution
- Data dependencies
  - The output of an instruction is the input of a later instruction
  - May result in data hazard if the later instruction that consumes the result is still in the pipeline

https://www.pollev.com/hungweitseng close in 1:30

How many data dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?

```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```

```
int temp = *a;
*a = *b;
*b = temp;
```

A. 1
B. 2
C. 3
D. 4
E. 5

How many dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?

```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```

```
int temp = *a;
*a = *b;
*b = temp;
```

A. 1
B. 2
C. 3
D. 4
E. 5

https://www.pollev.com/hungweitseng close in 1:30

How many data dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?

```
movl    (%rdi), %eax
xorl    (%rsi), %eax
movl    %eax, (%rdi)
xorl    (%rsi), %eax
movl    %eax, (%rsi)
xorl    %eax, (%rdi)
```

```
*a ^= *b;
*b ^= *a;
*a ^= *b;
```

A. 1
B. 2
C. 3
D. 4
E. 5

How many dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?

```
movl    (%rdi), %eax
xorl    (%rsi), %eax
movl    %eax, (%rdi)
xorl    (%rsi), %eax
movl    %eax, (%rsi)
xorl    %eax, (%rdi)
```

```
*a ^= *b;
*b ^= *a;
*a ^= *b;
```

A. 1
B. 2
C. 3
D. 4
E. 5

https://www.pollev.com/hungweitseng close in 1:30

Data hazards?

- How many pairs of data dependences in the following x86 instructions will result in data hazards if a memory operation (assume 100% cache hit rate) takes 4 cycles?

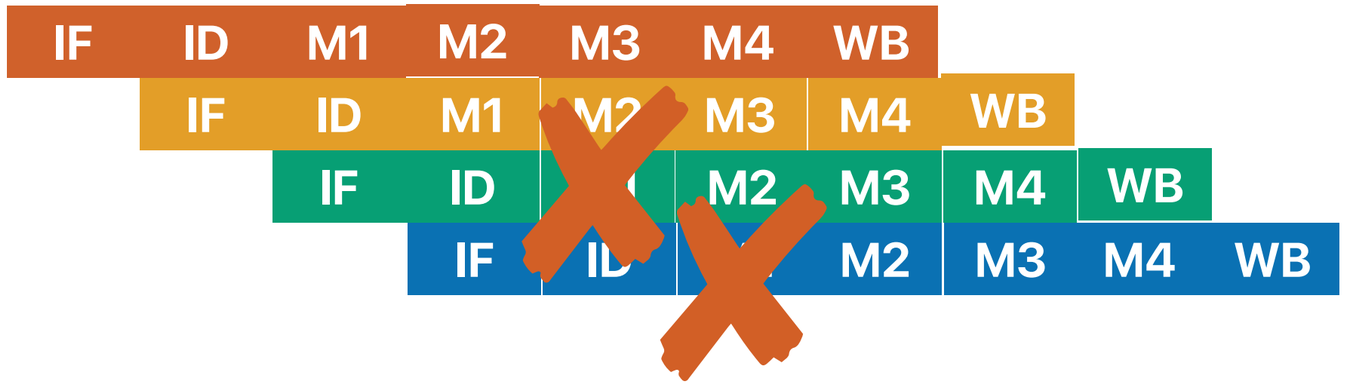
```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```

A. 0
B. 1
C. 2
D. 3
E. 4

Data hazards?

- How many pairs of data dependences in the following x86 instructions will result in data hazards if a memory operation (assume 100% cache hit rate) takes 4 cycles?

```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```



A. 0
B. 1
C. 2
D. 3
E. 4

Data hazards

```
① movl    (%rdi), %eax
② movl    (%rsi), %edx
③ movl    %edx, (%rdi)
④ movl    %eax, (%rsi)
```

%eax does not have our desired value

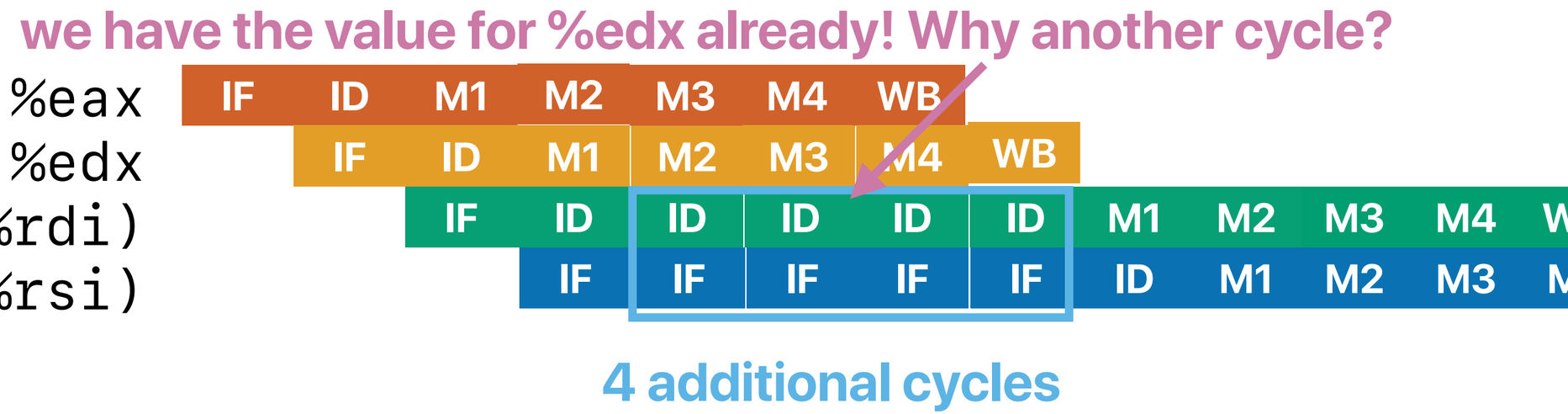
%edx does not have our desired value

	IF	ID	ALU/BR/M1	M2	M3	M4	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(4)	(3)	(2)	(1)			
5		(4)	(3)	(2)	(1)		
6			(4)	(3)	(2)	(1)	
7				(4)	(3)	(2)	(1)
8					(4)	(3)	(2)
9						(4)	(3)
10							(4)
11							
12							
13							
14							

Solution 1: Let's try "stall" again

- Whenever the input is not ready when the consumer is decoding, just stall — the consumer stays at ID.

```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```



Solution 1: Let's try "stall" again

- Whenever the input is not ready when the consumer is decoding, just stall — the consumer stays at ID.

```
① movl    (%rdi), %eax
② movl    (%rsi), %edx
③ movl    %edx, (%rdi)
④ movl    %eax, (%rsi)
```

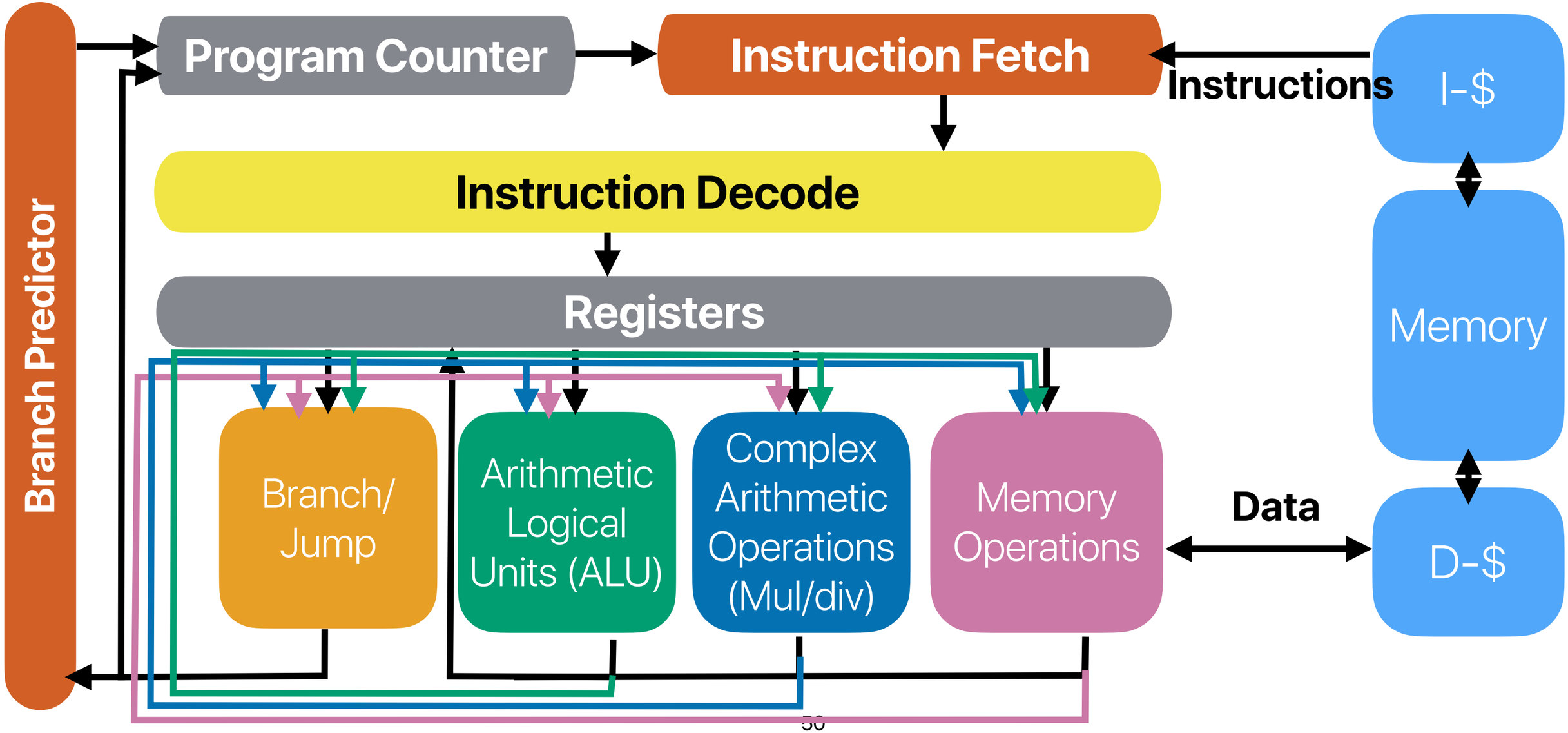
	IF	ID	ALU/BR/M1	M2	M3	M4	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(4)	(3)	(2)	(1)			
5	(4)	(3)		(2)	(1)		
6	(4)	(3)			(2)	(1)	
7	(4)	(3)				(2)	(1)
8	(4)	(3)					(2)
9		(4)	(3)				
10			(4)	(3)			
11				(4)	(3)		
12					(4)	(3)	
13						(4)	(3)
14							(4)

we have the value for %edx already!
Why another cycle?

Solution 2: Data forwarding

- Add logics/wires to forward the desired values to the demanding instructions

Data "forwarding"



https://www.pollev.com/hungweitseng close in 1:30

How many data dependencies are still problematic?

- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if a memory operation (assume 100% cache hit rate) takes 4 cycles?

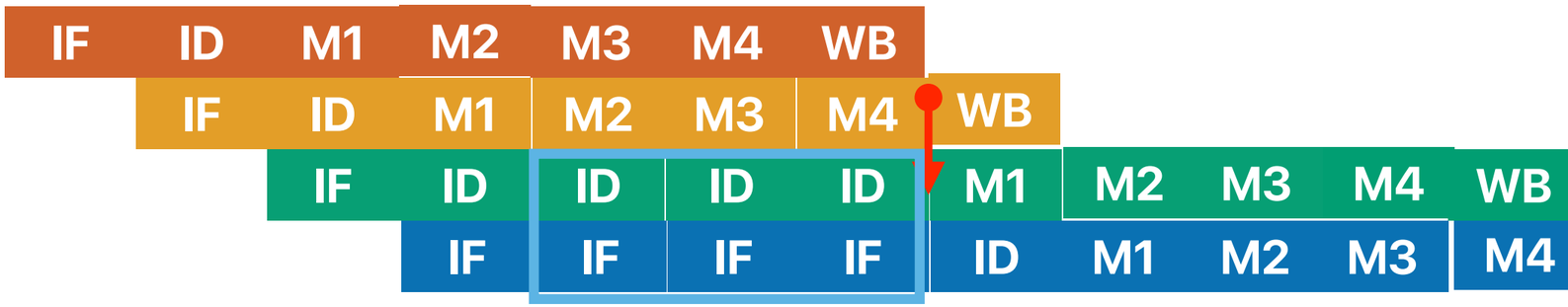
```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```

A. 0
B. 1
C. 2
D. 3
E. 4

How many data dependencies are still problematic?

- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if a memory operation (assume 100% cache hit rate) takes 4 cycles?

```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```



3 additional cycles

A. 0
B. 1
C. 2
D. 3
E. 4

```
int temp = *a;
*a = *b;
*b = temp;
```

Solution 2: Data forwarding

- Whenever the input is not ready when the consumer is decoding, just stall — the consumer stays at ID.

```
① movl    (%rdi), %eax
② movl    (%rsi), %edx
③ movl    %edx, (%rdi)
④ movl    %eax, (%rsi)
```

	IF	ID	ALU/BR/M1	M2	M3	M4	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(4)	(3)	(2)	(1)			
5	(4)	(3)		(2)	(1)		
6	(4)	(3)			(2)	(1)	
7	(4)	(3)	data forwarding			(2)	(1)
8		(4)	(3)				(2)
9			(4)	(3)			
10				(4)	(3)		
11					(4)	(3)	
12						(4)	(3)
13							(4)
14							

https://www.pollev.com/hungweitseng close in 1:30

How many of data hazards w/ Data Forwarding?

- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if both a memory operation and an xorl take 4 cycles?

```
① movl    (%rdi), %eax
② xorl    (%rsi), %eax
③ movl    %eax, (%rdi)
④ xorl    (%rsi), %eax
⑤ movl    %eax, (%rsi)
⑥ xorl    %eax, (%rdi)
```

```
*a ^= *b;
*b ^= *a;
*a ^= *b;
```

A. 0
B. 1
C. 2
D. 3
E. 4

How many of data hazards w/ Data Forwarding?

- How many pairs of data dependences in the following x86 instructions are still problematic with data forwarding if both a memory operation and an xorl take 4 cycles?

```
① movl    (%rdi), %eax
② xorl    (%rsi), %eax
③ movl    %eax, (%rdi)
④ xorl    (%rsi), %eax
⑤ movl    %eax, (%rsi)
⑥ xorl    %eax, (%rdi)
```

A. 0
B. 1
C. 2
D. 3
E. 4

	IF	ID	ALU/BR/M1	M2	M3	M4/XORL	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(3)	(2)		(1)			
5	(3)	(2)			(1)		
6	(3)	(2)				(1)	
7	(4)	(3)	(2)				(1)
8	(4)	(3)		(2)			
9	(4)	(3)			(2)		
10	(4)	(3)				(2)	
11	(5)	(4)	(3)				(2)
12	(6)	(5)	(4)	(3)			
13	(6)	(5)		(4)	(3)		
14	(6)	(5)			(4)	(3)	
15	(6)	(5)				(4)	(3)
16		(6)	(5)				(4)
17			(6)	(5)			
18				(6)	(5)		
19					(6)	(5)	
20						(6)	(5)
21							(6)
22							

Let's extend the example a bit...

```
for(i = 0; i < count; i++) {
    int64_t temp = a[i];
    a[i] = b[i];
    b[i] = temp;
}
```

```
.L9:
① movq    (%rdi,%rax), %rsi
② movq    (%rcx,%rax), %r8
③ movq    %r8, (%rdi,%rax)
④ movq    %rsi, (%rcx,%rax)
⑤ addq    $8, %rax
⑥ cmpq    %r9, %rax
⑦ jne     .L9
⑧ movq    (%rdi,%rax), %rsi
⑨ movq    (%rcx,%rax), %r8
⑩ movq    %r8, (%rdi,%rax)
⑪ movq    %rsi, (%rcx,%rax)
⑫ addq    $8, %rax
⑬ cmpq    %r9, %rax
⑭ jne     .L9
```

	IF	ID	ALU/BR/M1	M2	M3	M4/XORL	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(4)	(3)	(2)	(1)			
5	(4)	(3)		(2)	(1)		
6	(4)	(3)			(2)	(1)	
7	(4)	(3)				(2)	(1)
8	(5)	(4)	(3)				(2)
9	(6)	(5)	(4)	(3)			
10	(7)	(6)	(5)	(4)	(3)		
11	(8)	(7)	(6)	(5)	(4)	(3)	
12	(9)	(8)	(7)	(6)	(5)	(4)	(3)
13	(10)	(9)	(8)	(7)	(6)	(5)	(4)
14	(11)	(10)	(9)	(8)	(7)	(6)	(5)
15	(11)	(10)		(9)	(8)	(7)	(6)
16	(11)	(10)			(9)	(8)	(7)
17	(11)	(10)				(9)	(8)
18	(12)	(11)	(10)				(9)
19	(13)	(12)	(11)	(10)			
20	(14)	(13)	(12)	(11)	(10)		
21		(14)	(13)	(12)	(11)	(10)	
22			(14)	(13)	(12)	(11)	(10)

10 cycles for 7 instructions

CPI = 1.43

https://www.pollev.com/hungweitseng close in 1:30

The effect of code optimization

- By reordering which pair of the following instruction stream can we reduce stalls without affecting the correctness of the code?

```
① movq    (%rdi,%rax), %rsi
② movq    (%rcx,%rax), %r8
③ movq    %r8, (%rdi,%rax)
④ movq    %rsi, (%rcx,%rax)
⑤ addq    $8, %rax
⑥ cmpq    %r9, %rax
⑦ jne     .L9
```

A. (1) & (2)

B. (2) & (3)

C. (3) & (5)

D. (4) & (6)

E. No ordering can help reduce the stalls

The effect of code optimization

- By reordering which pair of the following instruction stream can we reduce stalls without affecting the correctness of the code?

```
① movq    (%rdi,%rax), %rsi
② movq    (%rcx,%rax), %r8
③ movq    %r8, (%rdi,%rax)
④ movq    %rsi, (%rcx,%rax)
⑤ addq    $8, %rax
⑥ cmpq    %r9, %rax
⑦ jne     .L9
```

A. (1) & (2)

B. (2) & (3)

C. (3) & (5)

D. (4) & (6)

E. No ordering can help reduce the stalls

Compiler optimization

```
for(i = 0; i < count; i++) {
    int64_t temp = a[i];
    a[i] = b[i];
    b[i] = temp;
}
```

```
movq    (%rdi,%rax), %rsi
movq    (%rcx,%rax), %r8
movq    %r8, (%rdi,%rax)
movq    %rsi, (%rcx,%rax)
addq    $8, %rax
cmpq    %r9, %rax
jne     .L9
movq    (%rdi,%rax), %rsi
movq    (%rcx,%rax), %r8
movq    %r8, (%rdi,%rax)
movq    %rsi, (%rcx,%rax)
addq    $8, %rax
cmpq    %r9, %rax
jne     .L9
```

```
.L9:
①  movq    (%rcx,%rax), %r8
②  movq    (%rdi,%rax), %rsi
③  movq    %r8, (%rdi,%rax)
④  movq    %rsi, (%rcx,%rax)
⑤  addq    $8, %rax
⑥  cmpq    %r9, %rax
⑦  jne     .L9
⑧  movq    (%rcx,%rax), %r8
⑨  movq    (%rdi,%rax), %rsi
⑩  movq    %r8, (%rdi,%rax)
⑪  movq    %rsi, (%rcx,%rax)
⑫  addq    $8, %rax
⑬  cmpq    %r9, %rax
⑭  jne     .L9
```

	IF	ID	ALU/BR/M1	M2	M3	M4/XORL	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(4)	(3)	(2)	(1)			
5	(4)	(3)		(2)	(1)		
6	(4)	(3)			(2)	(1)	
7	(5)	(4)	(3)			(2)	(1)
8	(6)	(5)	(4)	(3)			(2)
9	(7)	(6)	(5)	(4)	(3)		
10	(8)	(7)	(6)	(5)	(4)	(3)	
11	(9)	(8)	(7)	(6)	(5)	(4)	(3)
12	(10)	(9)	(8)	(7)	(6)	(5)	(4)
13	(11)	(10)	(9)	(8)	(7)	(6)	(5)
14	(11)	(10)		(9)	(8)	(7)	(6)
15	(11)	(10)			(9)	(8)	(7)
16	(12)	(11)	(10)			(9)	(8)
17	(13)	(12)	(11)	(10)			(9)
18	(14)	(13)	(12)	(11)	(10)		
19		(14)	(13)	(12)	(11)	(10)	
20			(14)	(13)	(12)	(11)	(10)
21				(14)	(13)	(12)	(11)
22					(14)	(13)	(12)

9 cycles for 7 instructions

CPI = 1.29

Missing opportunities

```
for(i = 0; i < count; i++) {
    int64_t temp = a[i];
    a[i] = b[i];
    b[i] = temp;
}
```

```
:
movq    (%rcx,%rax), %r8
movq    (%rdi,%rax), %rsi
movq    %r8, (%rdi,%rax)
movq    %rsi, (%rcx,%rax)
addq    $8, %rax
cmpq    %r9, %rax
jne     .L9
movq    (%rcx,%rax), %r8
movq    (%rdi,%rax), %rsi
movq    %r8, (%rdi,%rax)
movq    %rsi, (%rcx,%rax)
addq    $8, %rax
cmpq    %r9, %rax
jne     .L9
```

```
.L9:
①  movq    (%rcx,%rax), %r8
②  movq    (%rdi,%rax), %rsi
③  movq    %r8, (%rdi,%rax)
④  movq    %rsi, (%rcx,%rax)
⑤  addq    $8, %rax
⑥  movq    (%rcx,%rax), %r8
⑦  movq    (%rdi,%rax), %rsi
⑧  cmpq    %r9, %rax
⑨  jne     .L9
⑩  movq    %r8, (%rdi,%rax)
⑪  movq    %rsi, (%rcx,%rax)
⑫  addq    $8, %rax
⑬  cmpq    %r9, %rax
⑭  jne     .L9
```

	IF	ID	ALU/BR/M1	M2	M3	M4/XORL	WB
1	(1)						
2	(2)	(1)					
3	(3)	(2)	(1)				
4	(4)	(3)	(2)	(1)			
5	(4)	(3)		(2)	(1)		
6	(4)	(3)			(2)	(1)	
7	(5)	(4)	(3)			(2)	(1)
8	(6)	(5)	(4)	(3)			(2)
9	(7)	(6)	(5)	(4)	(3)		
10	(8)	(7)	(6)	(5)	(4)	(3)	
11	(9)	(8)	(7)	(6)	(5)	(4)	(3)
12	(10)	(9)	(8)	(7)	(6)	(5)	(4)
13	(11)	(10)	(9)	(8)	(7)	(6)	(5)
14	(12)	(11)	(10)	(9)	(8)	(7)	(6)
15	(13)	(12)	(11)	(10)	(9)	(8)	(7)
16	(14)	(13)	(12)	(11)	(10)	(9)	(8)
17		(14)	(13)	(12)	(11)	(10)	(9)
18			(14)	(13)	(12)	(11)	(10)
19				(14)	(13)	(12)	(11)
20					(14)	(13)	(12)
21						(14)	(13)
22							(14)

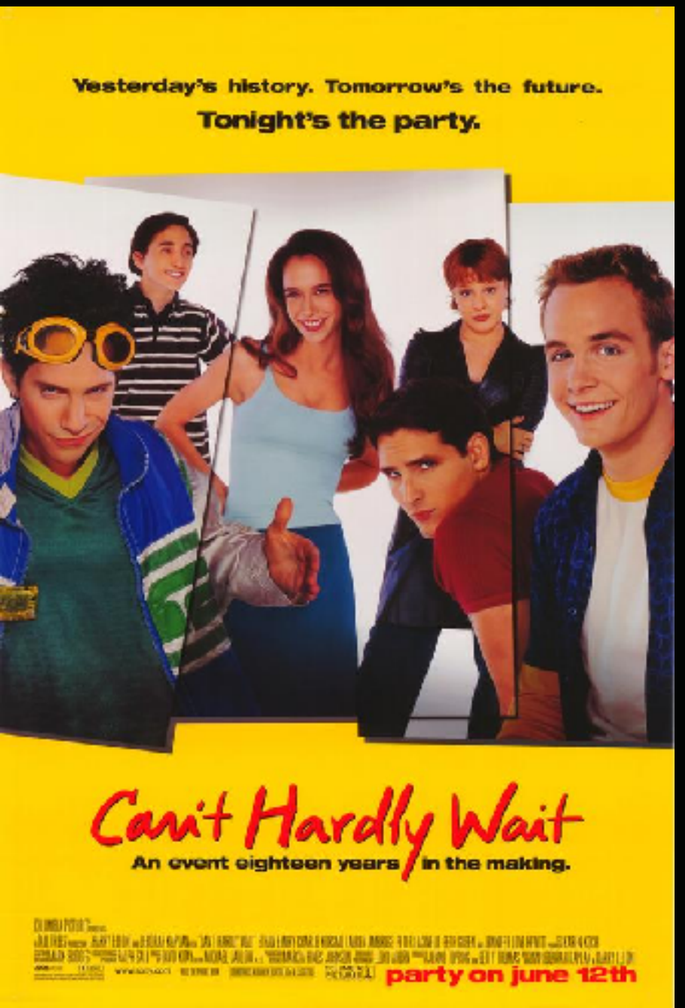
7 cycles for 7 instructions

CPI = 1

Compiler cannot optimize across branch since it's predicted during runtime!

Computer
Science &
Engineering

203



つづく