- Which of the following is generally true about ISAs?

  A. Many models of processors can support one ISA

  B. An ISA is unique to one model of processor

  C. Every processor can support multiple ISAs

  D. Each processor manufacturer has its own ISA

  E. None of the above

- For the C code below:

```
if(i < j)
    i++;
```

Assume `$t0 = i = 2`
`$t1 = j = 4`

Which of the following is the correct translation in MIPS?
Assume that `$t0` has `i` and `$t1` has `j`.
(`slt rd,rs,rt` does: `R[rd]=1` if `R[rs] < R[rt]`, else `R[rd]=0`)

A:
```
$t2 = 1                slt $t2, $t0, $t1
$t2 != 0, jump to false bne $t2, $zero, false
                       addi $t0, $t0, 1
false: next instruction
```

B:
```
$t2 = 0                slt $t2, $t1, $t0
                       bne $t2, $zero, true
true:                  addi $t0, $t0, 1
                       next instruction
```
$t2 == 0, execute addi, but it goes to true anyway...

C:
```
$t2 = 0                slt $t2, $t1, $t0
$t2 == 0, jump to false beq $t2, $zero, false
                       addi $t0, $t0, 1
false: next instruction
```

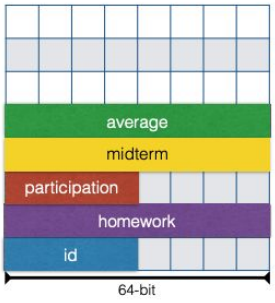
D:
```
                       $t2 = 1
                       slt $t2, $t0, $t1
                       beq $t2, $zero, false
                       addi $t0, $t0, 1
false: next instruction
```
$t2 == 1, execute addi

E: none of the above

- Which of the following is NOT correct about these two ISAs?
  - A. x86 provides more instructions than MIPS
  - B. x86 usually needs more instructions to express a program
  - C. An x86 instruction may access memory for 3 times
  - D. An x86 instruction may be shorter than a MIPS instruction
  - E. An x86 instruction may be longer than a MIPS instruction

- Consider the following data structure:

```
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```




What's the output of `printf("%lu\n",sizeof(struct student))`?

  - A. 20
  - B. 28
  - C. 32
  - D. 36
  - E. 40

- Consider the following performance metrics
  1. Network Bandwidth (data/sec)
  2. End-to-end Latency (ms)
  3. Frame Rate (frames/sec)
  4. Throughput (ops/sec)

  Which option contains the best match of the most important performance metric for each application?

| | WoW or LOL (Online gaming) | Halo (FPS) | Torrent download | Google Server Farm |
|---|---|---|---|---|
| A | 4 | 3 | 1 | 2 |
| B | 4 | 1 | 3 | 2 |
| C | 2 | 1 | 3 | 4 |
| D | 2 | 3 | 1 | 4 |
| E | None of the above | | | |

- Assume that we have an application composed with a total of 500000 instructions, in which 20% of them are the load/store instructions with an average CPI of 6 cycles, and the rest instructions are integer instructions with average CPI of 1 cycle. If the processor runs at 2 GHz, how long is the execution time?

A. 500000 ns
B. 1000000 ns
C. 2000000 ns
D. 3500000 ns
E. None of the above

- Assume that we have an application composed with a total of 500000 instructions, in which 20% of them are the load/store instructions with an average CPI of 6 cycles, and the rest instructions are integer instructions with average CPI of 1 cycle.
  - If we double the CPU clock rate to 4GHz but keep using the same memory module, the average CPI for load/store instruction will become 12 cycles. What's the performance improvement after this change?

A. No change
B. 1.25
C. 1.5
D. 2
E. None of the above


Sysbench 2014 from http://www.anandtech.com/

Why does an Intel Core i7 @ 3.5 GHz usually perform better than an Intel Core i5 @ 3.5 GHz or AMD FX-8350@4GHz?

A. Because the instruction count of the program are different
B. Because the clock rate of AMD FX is higher
C. Because the CPI of Core i7 is better
D. Because the clock rate of AMD FX is higher and CPI of Core i7 is better
E. None of the above

# Example of Amdahl's Law


- Call of Duty Black Ops II loads a zombie map for 10 minutes on my current machine, and spends **20%** of this time in integer instructions
- How much faster must you make the integer unit to make the map loading **1 minute** faster?

A. 1.11
B. 1.25
C. 1.31
D. 2.00
E. 2.51

$$\text{Speedup} = \frac{1}{\frac{x}{s} + (1-x)}$$

$$\text{Speedup} = \frac{10}{10-1} = 1.111$$

$$1.111 = \frac{1}{\frac{20\%}{s} + (1-20\%)}$$

$$S = 2$$

# Example of Amdahl's Law


- Call of Duty Black Ops II loads a zombie map for 10 minutes on my current machine, and spends **20%** of this time in integer instructions
- How much faster must you make the integer unit to make the map loading **5 minutes** faster?

A. 0.66x
B. 16.6x
C. 66.6x
D. 100x
E. None of the above

$$\text{Speedup} = \frac{10}{10-5} = 2$$

$$\text{Speedup} = \frac{1}{\frac{x}{s} + (1-x)}$$

$$2 = \frac{1}{\frac{20\%}{s} + (1-20\%)}$$

$$S = -0.66$$


- Call of Duty Black Ops II loads a zombie map for 10 minutes on my current machine.
- It spends **20%** of loading map time in **integer ALU operations**
- It spends **35%** of loading map time in **accessing SSD**
- If I have $200 to upgrade the system, should I:

A. Upgrading my CPU to speed up the integer instruction processing by 2x

B. Replacing my SSD with a high-end model that reduces the access time from 20us to 12us


- Call of Duty Black Ops II loads a zombie map for 10 minutes on my current machine.
- It spends **20%** of loading map time in **integer ALU operations**
- It spends **35%** of loading map time in **accessing SSD**
- If I have $200 to upgrade the system, should I:

| Replacing CPU | Replacing SSD |
|---|---|
| $\text{Speedup} = \frac{1}{\frac{x}{s} + (1-x)}$ <br> $1.11 = \frac{1}{\frac{20\%}{2} + (1-20\%)}$ | $\text{Speedup} = \frac{1}{\frac{x}{s} + (1-x)}$ <br> $1.16 = \frac{1}{\frac{35\%}{20/12} + (1-35\%)}$ |

# Add cores or features?

- Recent advances in process technology have quadruple the number transistors you can fit on your die.
- Currently, your key customer can use up to 4 processors for 40% of their application.
- Which will you choose?
  - A. Increase the number of processors from 1 to 4
  - B. Use 2 cores, but add features that will allow the application to use two cores for 80% of execution.

$$S_{quad\text{-}core} = \frac{1}{\frac{x}{s} + (1\text{-}x)} \qquad 1.43 = \frac{1}{\frac{40\%}{4} + (1\text{-}40\%)}$$

$$S_{dual\text{-}core} = \frac{1}{\frac{x}{s} + (1\text{-}x)} \qquad 1.67 = \frac{1}{\frac{80\%}{2} + (1\text{-}80\%)}$$



- Assume that memory access takes 30% of execution time.
  - Cache can speedup 80% of memory operation by a factor of 4
  - L2 cache can speedup 50% of the remaining 20% by a factor of 2
- What's the total speedup?
  - A. 1.22
  - B. 1.23
  - C. 1.24
  - D. 2.63
  - E. 2.86

Execution time can be optimized by L1 only = 30%*80% = 24%

Execution time can be optimized by L2 only = 30%*50%*20% = 3%

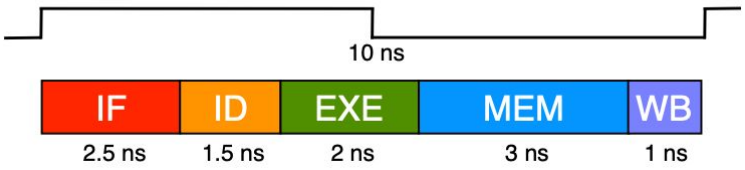
$$\text{Speedup} = \frac{1}{(1 - 0.27) + \frac{0.24}{4} + \frac{0.03}{2}} = 1.24$$

- Regarding power and energy, how many of the following statements are correct?
  - ① Lowering the power consumption helps extending the battery life
  - ② Lowering the power consumption helps reducing the heat generation
  - ③ Lowering the energy consumption helps reducing the electricity bill
  - ④ A CPU with 10% utilization can still consume 33% of the peak power
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

- If we are able to cram more transistors within the same chip area (Moore's law continues), but the power consumption per transistor remains the same. Right now, if we power the chip with the same power consumption but put more transistors in the same area because the technology allows us to. How many of the following statements are true?
  - ① The power consumption per chip will increase
  - ② The power density of the chip will increase
  - ③ Given the same power budget, we may not able to power on all chip area if we maintain the same clock rate
  - ④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

- How many of the following statements about a single-cycle processor is correct?
  - ① The CPI of a single-cycle processor is always 1
  - ② If the single-cycle implements MIPS ISA, the memory instruction will determine the cycle time
  - ③ Hardware elements are mostly idle during a cycle
  - ④ We can always reduce the cycle time of a single-cycle processor by supporting fewer instructions — Only if this instruction is the most time-critical one
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

- The following diagram shows the latency in each part of a single-cycle processor:




If we can make each part as a "pipeline stage", what's the maximum speedup we can achieve? (choose the closest one)

A. 3.33
B. 4
C. 5
D. 6.67
E. 10

$$\text{Speedup} = \frac{\#\_of\_ins * 1 * 10ns}{\#\_of\_ins\_* 1 * 3ns}$$

- How many of the following descriptions about pipelining is correct?
  ① You can always divide stages into short stages with latches to improve performance
  — Only if this stage is the most time-critical one
  ② Pipeline registers incur overhead for each pipeline stage
  ③ The latency of executing an instruction in a pipeline processor is longer than a single-cycle processor
  ④ The throughput of a pipeline processor is usually better than a single-cycle processor
  — You have pipeline registers and each stage needs to be equally long

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

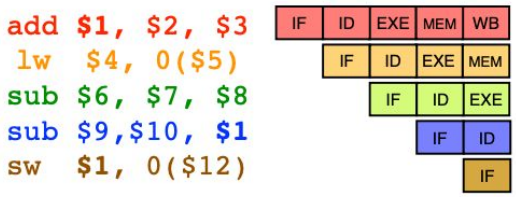
- Given the current 5-stage pipeline,


how many of the following MIPS code can work correctly (i.e. generate the same result as a single-cycle processor)?

| | I | II | III | IV |
|---|---|---|---|---|
| a: | `add $1, $2, $3` | `add $1, $2, $3` | `add $1, $2, $3` | `add $1, $2, $3` |
| b: | `lw $4, 0($1)` | `lw $4, 0($5)` | `lw $4, 0($5)` | `lw $4, 0($5)` |
| c: | `sub $6, $7, $8` | `sub $6, $7, $8` | `bne $0, $7, L` | `sub $6, $7, $8` |
| d: | `sub $9,$10,$11` | `sub $9, $1, $10` | `sub $9,$10,$11` | `sub $9,$10,$11` |
| e: | `sw $1, 0($12)` | `sw $11, 0($12)` | `sw $1, 0($12)` | `sw $1, 0($12)` |
| | b cannot get $1 produced by a before WB | both a and d are accessing $1 at 5th cycle | We don't know if d & e will be executed or not until c finishes | |

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

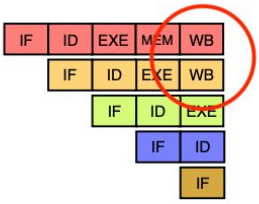
- What just happened here is problematic if we change one of the source register of the 2nd sub instruction?




A. The register file is trying to read and write at the same cycle
B. The ALU and data memory are both active at the same cycle
C. A value is used before it's produced
D. Both A and B
E. Both A and C

- What pair of instructions will be problematic if we allow R-type instructions to skip the “MEM” stage?

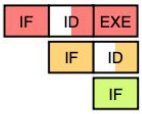
```
a: lw  $1, 0($2)
b: add $3, $4, $5
c: sub $6, $7, $8
d: sub $9,$10,$11
e: sw  $1, 0($12)
```




A. a & b
B. a & c
C. b & e
D. c & e
E. None

- What just happened here is problematic for the following instructions in our current pipeline?

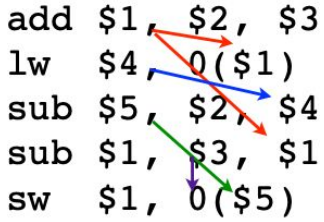
```
add $1, $2, $3
 lw  $4, 0($1)
sub $6, $7, $8
sub $9,$10, $1
sw  $1, 0($12)
```




A. The register file and memory are both active at the same cycle
B. The ALU and data memory are both active at the same cycle
C. A value is used before it’s produced
D. Both A and B
E. Both A and C

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

```
add $1, $2, $3
lw  $4, 0($1)
sub $5, $2, $4
sub $1, $3, $1
sw  $1, 0($5)
```



**No every "data dependency" will lead to "data hazards".**

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

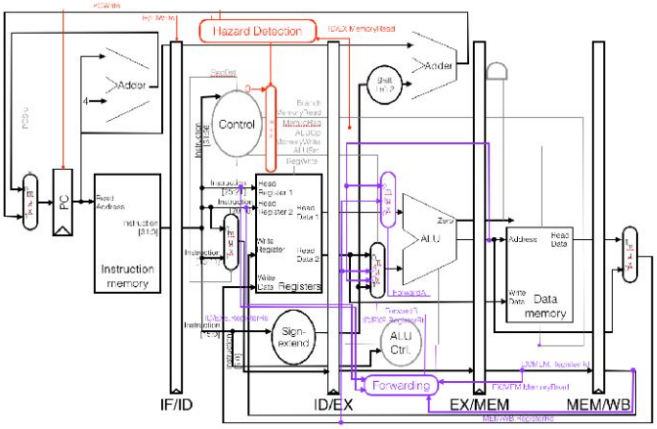
- By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

① `add $1, $2, $3`
② `lw  $4, 0($1)`
③ `sub $5, $2, $4`
④ `sub $1, $3, $1`
⑤ `sw  $1, 0($5)`

A. (1) & (2)
B. (2) & (3)
C. (3) & (4)
D. (4) & (5)
E. (3) & (5)

- Consider the following code and the pipeline we designed

```
LOOP: lw   $t3, 0($s0)
      addi $t0, $t0, 1
      add  $v0, $v0, $t3
      addi $s0, $s0, 4
      bne  $t1, $t0, LOOP
      sw   $v0, 0($s1)
```



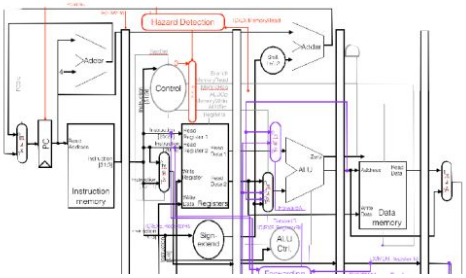

How many cycles does the processor need to stall before we figure out the next instruction after "`bne`"?

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

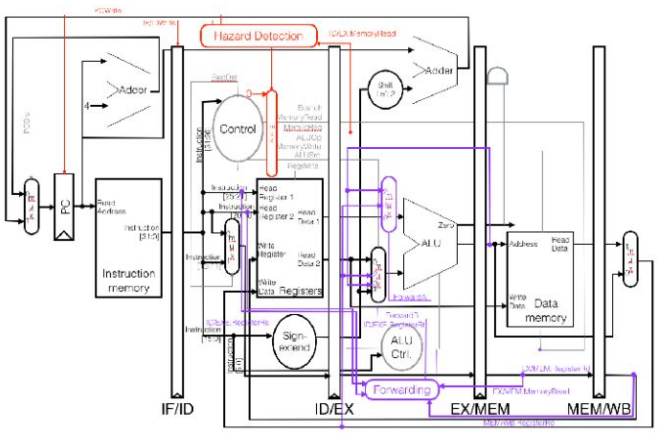
- How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor

① The target address when branch is taken is not available for instruction fetch stage of the next cycle
② The target address when branch is not-taken is not available for instruction fetch stage of the next cycle
③ The branch outcome cannot be decided until the comparison result of ALU is out
④ The next instruction needs the branch instruction to write back its result

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

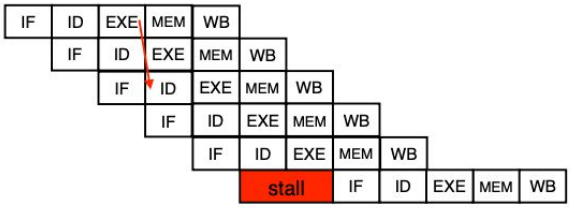


- Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards, what's the average CPI if we execute this program on the 5-stage MIPS pipeline?
  - A. 1
  - B. 1.2
  - C. 1.4
  - D. 1.6
  - E. 1.8



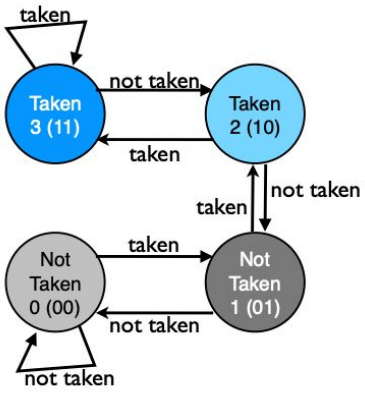

- The processor cannot determine the next PC to fetch

```
LOOP: lw   $t3, 0($s0)
      addi $t0, $t0, 1
      add  $v0, $v0, $t3
      addi $s0, $s0, 4
      bne  $t1, $t0, LOOP
      sw   $v0, 0($s1)
```




7 cycles per loop

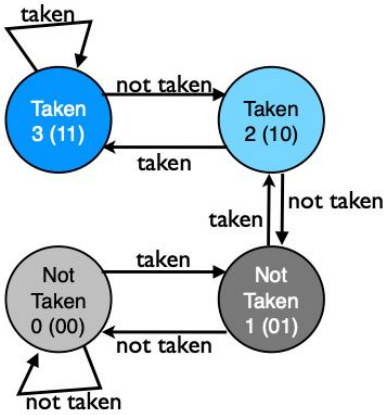
```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```




| i | branch? | state | prediction | actual |
|---|---|---|---|---|
| 0 | X | 00 | NT | T |
| 0 | Y | 00 | NT | T |
| 1 | X | 01 | NT | NT |
| 1 | Y | 01 | NT | T |
| 2 | X | 00 | NT | T |
| 2 | Y | 10 | T | T |
| 3 | X | 01 | NT | NT |
| 3 | Y | 11 | NT | T |
| 4 | X | 00 | NT | T |
| 4 | Y | 10 | T | T |
| 5 | X | 01 | NT | NT |
| 5 | Y | 11 | NT | T |
| 6 | X | 00 | NT | T |
| 6 | Y | 10 | T | T |

**For branch Y, almost 100%,**
**For branch X, only 50%**

```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```




- What's the overall branch prediction (include both branches) accuracy for this nested for loop? (assume all states started with 00)
  - A. ~25%
  - B. ~33%
  - C. ~50%
  - D. ~67%
  - E. ~75%

```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
       a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```

| i | branch? | GHR | state | prediction | actual |
|---|---|---|---|---|---|
| 0 | X | 000 | 00 | NT | T |
| 0 | Y | 001 | 00 | NT | T |
| 1 | X | 011 | 00 | NT | NT |
| 1 | Y | 110 | 00 | NT | T |
| 2 | X | 101 | 00 | NT | T |
| 2 | Y | 011 | 00 | NT | T |
| 3 | X | 111 | 00 | NT | NT |
| 3 | Y | 110 | 01 | NT | T |
| 4 | X | 101 | 01 | NT | T |
| 4 | Y | 011 | 01 | NT | T |
| 5 | X | 111 | 00 | NT | NT |
| 5 | Y | 110 | 10 | T | T |
| 6 | X | 101 | 10 | T | T |
| 6 | Y | 011 | 10 | T | T |
| 7 | X | 111 | 00 | NT | NT |
| 7 | Y | 110 | 11 | T | T |
| 8 | X | 101 | 11 | T | T |
| 8 | Y | 011 | 11 | T | T |
| 9 | X | 111 | 00 | NT | NT |
| 9 | Y | 110 | 11 | T | T |
| 10 | X | 101 | 11 | T | T |
| 10 | Y | 011 | 11 | T | T |

Nearly perfect after this

- Why the performance is better when option is not “0”
  - ① The amount of dynamic instructions needs to execute is a lot smaller
  - ② The amount of branch instructions to execute is smaller
  - ③ The amount of branch mis-predictions is smaller
  - ④ The amount of data accesses is smaller

  A. 0

  B. 1

  C. 2

  D. 3

  E. 4

```
if(option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold) branch X
            sum ++;
    }
}
```

| | Without sorting | With sorting |
|---|---|---|
| The prediction accuracy of X before threshold | 50% | 100% |
| The prediction accuracy of X after threshold | 50% | 100% |

- How many of the following statements explains the reason why B outperforms A with compiler optimizations
  - ① B has lower dynamic instruction count than A
  - ② B has significantly lower branch mis-predictions than A
  - ③ B has significantly fewer branch instructions than A
  - ④ B can incur fewer data hazards
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

**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;
```

- 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 iterati
  - ② C has significantly lower branch mis-predictions 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 suppor
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

**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
     while(x)     {
          c += table[(x & 0xF)];
          x = x >> 4;
     }
     return 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 — potentially could be more, but aggressive compiler can do loop unrolling
  - ② D has significantly lower branch mis-predictions 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;
}
```

you know exactly the number of iterations → compiler can do loop unrolling!

- Assume your processor takes only 1 cycle to process an instruction if the DRAM is the same speed as the processor. In fact, the latency of DRAM is 100 cycles.
  - Ignore all virtual memory overheads and processor optimizations
  - The application contains 20% instructions that perform data memory accesses
  - How many cycles do you need to process an instruction on average?
  - A. ~ 10
  - B. ~ 20
  - C. ~ 100
  - D. ~ 120
  - E. ~ 200

instruction fetch, data access

average = 1 + 1*100 + 0.2*100 = 121

- Assume your processor takes only 1 cycle to process an instruction if the DRAM is the same speed as the processor. In fact, the latency of DRAM is 100 cycles.
  - Now, if we add a cache between CPU and DRAM. If 90% of time, the data/instruction can be found in the cache — no additional cycles is wasted to fetch the data/instruction.
  - If the data is missing in the cache, it takes 100 cycles to retrieve data from the DRAM
  - You may ignore all virtual memory overheads and processor optimizations
  - The application contains 20% instructions that perform data memory accesses
  - How many cycles do you need to process an instruction on average?

A. ~ 10
B. ~ 20
C. ~ 100
D. ~ 120
E. ~ 200

instruction fetch  data access

average = 1 + (1*100 + 0.2*100)*(1-90%) = 13

- Which description about locality of arrays `sum` and `A` in the following code is the most accurate?

```
for(i = 0; i< 100000; i++)
{
    sum[i%10] += A[i];
}
```

A. Access of `A` has temporal locality, `sum` has spatial locality
B. Both `A` and `sum` have temporal locality, and `sum` also has spatial locality
C. Access of `A` has spatial locality, `sum` has temporal locality
D. Both `A` and `sum` have spatial locality
E. Both `A` and `sum` have spatial locality, and `sum` also has temporal locality

- L1 data (D-L1) cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block
  - Assume 64-bit memory address

Which of the following is correct?

A. Tag is 49 bits
B. Index is 8 bits
C. Offset is 7 bits
D. The cache has 1024 sets
E. None of the above

$$C = ABS$$
$$64KB = 2 * 64 * S$$
$$S = 512$$
$$\text{offset} = \lg(64) = 6 \text{ bits}$$
$$\text{index} = \lg(512) = 9 \text{ bits}$$
$$\text{tag} = 64 - \lg(512) - \lg(64) = 49 \text{ bits}$$

- L1 data (D-L1) cache configuration of Core i7
  - Size 32KB, 8-way set associativity, 64B block
  - Assume 64-bit memory address
  - Which of the following is NOT correct?

A. Tag is 52 bits
B. Index is 6 bits
C. Offset is 6 bits
D. The cache has 128 sets

$$C = ABS$$
$$32KB = 8 * 64 * S$$
$$S = 64$$
$$\text{offset} = \lg(64) = 6 \text{ bits}$$
$$\text{index} = \lg(64) = 6 \text{ bits}$$
$$\text{tag} = 64 - \lg(64) - \lg(64) = 52 \text{ bits}$$

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.
  - Cache performance for the following code?

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

  - What's the data cache miss rate for this code?

A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%

# AMD Phenom II

**100% miss rate!**

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 48-bit address.

C = ABS
64KB = 2 * 64 * S
S = 512
offset = lg(64) = 6 bits
index = lg(512) = 9 bits
tag = the rest bits

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
```

| | address in hex | address in binary (tag / index / offset) | tag | index | hit? miss? |
|---|---|---|---|---|---|
| load a[0] | 0x20000 | 10 0000 0000 0000 0000 | 0x4 | 0 | miss |
| load b[0] | 0x30000 | 11 0000 0000 0000 0000 | 0x6 | 0 | miss |
| store c[0] | 0x10000 | 1 0000 0000 0000 0000 | 0x2 | 0 | miss, evict 0x4 |
| load a[1] | 0x20004 | 10 0000 0000 0000 0100 | 0x4 | 0 | miss, evict 0x6 |
| load b[1] | 0x30004 | 11 0000 0000 0000 0100 | 0x6 | 0 | miss, evict 0x2 |
| store c[1] | 0x10004 | 1 0000 0000 0000 0100 | 0x2 | 0 | miss, evict 0x4 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| load a[15] | 0x2003C | 10 0000 0000 0011 1100 | 0x4 | 0 | miss, evict 0x6 |
| load b[15] | 0x3003C | 11 0000 0000 0011 1100 | 0x6 | 0 | miss, evict 0x2 |
| store c[15] | 0x1003C | 1 0000 0000 0011 1100 | 0x2 | 0 | miss, evict 0x4 |
| load a[16] | 0x20040 | 10 0000 0000 0100 0000 | 0x4 | 1 | miss |
| load b[16] | 0x30040 | 11 0000 0000 0100 0000 | 0x6 | 1 | miss |
| store c[16] | 0x10040 | 1 0000 0000 0100 0000 | 0x2 | 1 | miss, evict 0x4 |

- D-L1 Cache configuration of intel Core i7 processor
  - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.
  - Cache performance for the following code?

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++) {
    c[i] = a[i] + b[i];
    //load a, b, and then store to c
}
```

  - What's the data cache miss rate for this code?

A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%

C = ABS
32KB = 8 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = 64 - lg(64) - lg(64) = 52 bits

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}
```

| | address | tag | index | ? |
|---|---|---|---|---|
| load a[0] | 0x20000 | 0x20 | 0 | miss |
| load b[0] | 0x30000 | 0x30 | 0 | miss |
| store c[0] | 0x10000 | 0x10 | 0 | miss |
| load a[1] | 0x20004 | 0x20 | 0 | hit |
| load b[1] | 0x30004 | 0x30 | 0 | hit |
| store c[1] | 0x10004 | 0x10 | 0 | hit |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| load a[15] | 0x2003C | 0x20 | 0 | hit |
| load b[15] | 0x3003C | 0x30 | 0 | hit |
| store c[15] | 0x1003C | 0x10 | 0 | hit |
| load a[16] | 0x20040 | 0x20 | 1 | miss |
| load b[16] | 0x30040 | 0x30 | 1 | miss |
| store c[16] | 0x1003C | 0x10 | 1 | miss |

**32*3/(512*3) = 1/16 = 6.25% (93.75% hit rate!)**

- 5-stage MIPS processor.
  - Application: 80% ALU, 20% Loads
  - L1 I-cache miss rate: 5%, hit time: 1 cycle
  - L1 D-cache miss rate: 10%, hit time: 1 cycle
  - L2 U-Cache miss rate: 20%, hit time: 10 cycles
  - Main memory hit time: 100 cycles
  - Assume the program is read only (nothing dirty)
  - What's the average CPI?

A. 1.1
B. 1.6
C. 2.1
D. 3.1
E. none of the above

$CPI_{Average} = CPI_{base} + miss\_rate * miss\_penalty$

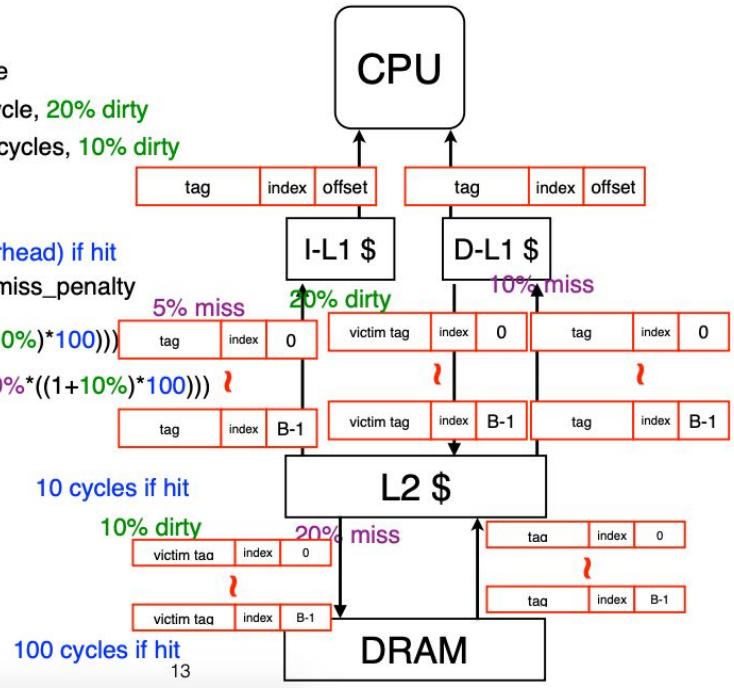
$= 1 + 100\% * (5\% * (10 + 20\% * (1 * 100)))$
$+ 20\% * (10\% * (1) * (10 + 20\% * ((1) * 100)))$
$= 3.1$

- Application: 80% ALU, 20% Load/Store
- L1 I-cache miss rate: 5%, hit time: 1 cycle
- L1 D-cache miss rate: 10%, hit time: 1 cycle, 20% dirty
- L2 U-Cache miss rate: 20%, hit time: 10 cycles, 10% dirty
- Main memory hit time: 100 cycles
- What's the average CPI?

1 cycle (no overhead) if hit

$CPI_{Average} = CPI_{base} + miss\_rate * miss\_penalty$

$= 1 + 100\% * (5\% * (10 + 20\% * ((1 + 10\%) * 100)))$
$+ 20\% * (10\% * (1 + 20\%) * (10 + 20\% * ((1 + 10\%) * 100)))$
$= 3.368$



- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.
  - Consider the following code
    - ```
      int a[16384], b[16384], c[16384];
      /* c = 0x10000, a = 0x20000, b = 0x30000 */
      for(i = 0; i < 512; i++) {
          c[i] = a[i] + b[i];
          //load a, b, and then store to c
      }
      ```
    - How many of the cache misses are “conflict misses”?

A. 6.25%
B. 66.67%
C. 68.75%
D. 93.75%
E. 100%

- D-L1 Cache configuration of Core i7
  - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, 32-bit OS?
  - Consider the following code?
    - ```
      int a[16384], b[16384], c[16384];
      /* c = 0x10000, a = 0x20000, b = 0x30000 */
      for(i = 0; i < 512; i++) {
          c[i] = a[i] + b[i];
          //load a, b, and then store to c
      }
      ```
    - How many of the cache misses are “compulsory misses”?

A. 6.25%
B. 33.33%
C. 66.67%
D. 68.75%
E. 100%

- Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity
  How many of the following are correct?
  ① Increasing associativity can reduce conflict misses
  ② Increasing associativity can reduce hit time
  ③ Increasing block size can increase the miss penalty
  ④ Increasing block size can reduce compulsory misses

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

Increases hit time because your data array is larger (longer time to fully charge your bit-lines)

You need to fetch more data for each miss

You bring more into the cache when a miss occurs

| i on row, j on col | i on col, j on row |
|---|---|
| `for(i = 0; i < ARRAY_SIZE; i++) { for(j = 0; j < ARRAY_SIZE; j++) { c[i][j] = a[i][j]+b[i][j]; } }` | `for(j = 0; j < ARRAY_SIZE; j++) { for(i = 0; i < ARRAY_SIZE; i++) { c[i][j] = a[i][j]+b[i][j]; } }` |

- What type(s) of cache locality does(do) the left-hand side code better exploit than the right-hand side code?

  A. Spatial locality
  B. Temporal locality
  C. Both localities
  D. None of them

# What data structure is performing better

| Array of objects | object of arrays |
|---|---|
| `struct grades`<br>`{`<br>`  int id;`<br>`  double *homework;`<br>`  double average;`<br>`};` | `struct grades`<br>`{`<br>`  int *id;`<br>`  double **homework;`<br>`  double *average;`<br>`};` |

- Considering your workload would like to calculate the average score of each homework, which data structure would deliver better performance?

  A. Array of objects

  B. Object of arrays

- Connecting architecture and software design now!
  - Block Algorithm for Matrix Multiplication
  - What value of n makes the block algorithm works the best?
  - If the demo machine has an L1 D-cache with 64KB, 2-way, 64B blocks, array_size is 1024, each word is "8 bytes"

  A. 16

  B. 32

  C. 64

  D. 128

  E. 256

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
        for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
          for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
            for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
              c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

- Consider the case when we run multiple instances of the given program at the same time, which pair of statements is correct?
  - ① The printed "address of a" is the same for every running instances
  - ② The printed "address of a" is different for each instance
  - ③ All running instances will print the same value of a
  - ④ Some instances will print the same value of a
  - ⑤ Each instance will print a different value of a

A. (1) & (3)

B. (1) & (4)

C. (1) & (5)

D. (2) & (3)

E. (2) & (4)

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int cpu, status;
    status = syscall(SYS_getcpu, &cpu, NULL, NULL);
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s process_nickname\n",argv[0]);
        exit(1);
    }
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and
address of a is %p\n",argv[1], cpu, a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %s is using CPU: %d. Value of a is %lf and
address of a is %p\n",argv[1], cpu, a, &a);
    return 0;
}
```

- If the L1 cache uses virtual memory address (virtual cache), how many of the following statements are correct?
  1. The processor can access the cache directly without translating memory addresses into physical addresses
  2. The TLB lookup can potentially occur concurrently with L1 cache access
  3. If two processes have the same virtual memory address, the virtual cache cannot distinguish which process does the cache block belong to
  4. If two virtual memory addresses map to the same physical location, there may exist multiple copies of that physical location

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

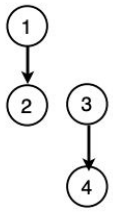
- Consider the following instructions:

```
1: lw   $t1, 0($a0)
2: add  $v0, $v0, $t1
3: addi $a0, $a0, 4
4: bne  $a0, $t0, LOOP
```

```
1: lw   $t1, 0($a0)
3: addi $a0, $a0, 4
2: add  $v0, $v0, $t1
4: bne  $a0, $t0, LOOP
```

Reordering which of the following pair of instructions would improve the performance without affecting correctness?

A. 1 and 3
B. 2 and 3
C. 2 and 4
D. 3 and 4
E. No room for optimizations



- The basic idea of SuperScalar is to duplicate the amount of functional units (e.g., ALUs) in the processor's pipeline to execute more than one instruction in each cycle. To achieve this goal, how many of the following modifications do we need in the processor architecture?

  ① Modifying the instruction fetch unit to fetch more instructions at each cycle

  We need to supply the pipeline with more than one instructions!

  ✗ Modifying the decode unit to parse more instructions and duplicate the number of registers to concurrently feed inputs to ALUs/pipeline registers

  You don't need more registers — but you do need to allow concurrent accesses of registers

  ③ Modifying the memory units to accept two requests at the same time.

  It's a must if you want to have two memory instructions in the pipeline

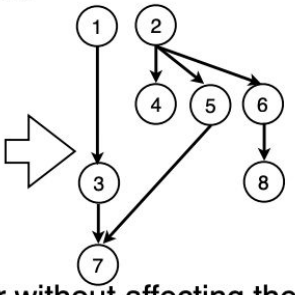
  ④ Modifying the data forwarding and hazard detection units

  Yes — you need to check more instructions in each pipeline stage

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

- Consider the following dynamic instructions:

```
1: lw   $t1, 0($a0)
2: addi $a0, $a0, 4
3: add  $v0, $v0, $t1
4: bne  $a0, $t0, LOOP
5: lw   $t1, 0($a0)
6: addi $a0, $a0, 4
7: add  $v0, $v0, $t1
8: bne  $a0, $t0, LOOP
```



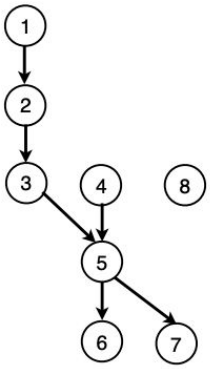

- Which of the following pair can we reorder without affecting the correctness if the branch prediction is perfect?

  A. 1 and 2
  B. 3 and 5
  C. 3 and 6
  D. 4 and 5
  E. 4 and 6

# False dependencies

- Consider the following dynamic instructions

```
1: lw   $t2, 0($a0)
2: add  $t2, $t0, $t2
3: sub  $t8, $t2, $t0
4: lw   $t2, 4($a0)
5: add  $t4, $t8, $t2
6: add  $t8, $t4, $t4
7: sw   $t4, 8($a0)
8: addi $a0, $a0, 4
```




which of the following pair is not a "false dependency"

| | |
|---|---|
| A. 1 and 4 | WAW |
| B. 1 and 8 | WAR |
| C. 5 and 7 | True dependency (RAW) |
| D. 4 and 8 | WAR |
| E. 7 and 8 | WAR |

# Register renaming

Original code

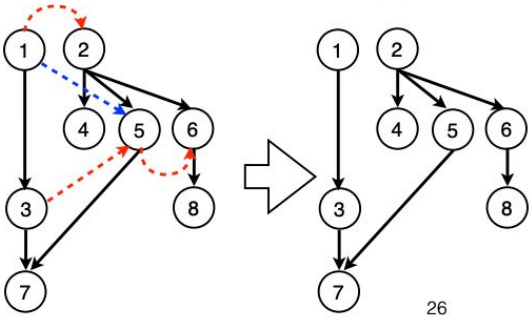
```
1: lw   $t1, 0($a0)
2: addi $a0, $a0, 4
3: add  $v0, $v0, $t1
4: bne  $a0, $t0, LOOP
5: lw   $t1, 0($a0)
6: addi $a0, $a0, 4
7: add  $v0, $v0, $t1
8: bne  $a0, $t0, LOOP
```

After renamed

```
1: lw   $p5 , 0($p1)
2: addi $p6 , $p1, 4
3: add  $p7 , $p4, $p5
4: bne  $p6 , $p2, LOOP
5: lw   $p8 , 0($p6)
6: addi $p9 , $p6, 4
7: add  $p10, $p7, $p8
8: bne  $p9 , $p2, LOOP
```

Register map

| cycle | $a0 | $t0 | $t1 | $v0 |
|---|---|---|---|---|
| 0 | p1 | p2 | p3 | p4 |
| 1 | p1 | p2 | p5 | p4 |
| 2 | p6 | p2 | p5 | p4 |
| 3 | p6 | p2 | p5 | p7 |
| 4 | p6 | p2 | p5 | p7 |
| 5 | p6 | p2 | p8 | p7 |
| 6 | p9 | p2 | p8 | p7 |
| 7 | p9 | p2 | p8 | p10 |
| 8 | p9 | p2 | p8 | p10 |



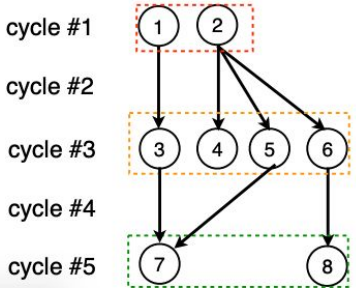
- Consider the following dynamic instructions

```
1: lw   $t1, 0($a0)
2: lw   $a0, 4($a0)
3: add  $v0, $v0, $t1
4: bne  $a0, $zero, LOOP
5: lw   $t1, 0($a0)
6: lw   $t2, 4($a0)
7: add  $v0, $v0, $t1
8: bne  $t2, $zero, LOOP
```

You code looks like this when performing "linked list" traversal

Assume a superscalar processor with unlimited issue width & physical registers that can fetch up to 4 instructions per cycle, 2 cycles to execute a memory instruction how many cycles it takes to issue all instructions?

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



- How many of the following would happen given the modern processor microarchitecture?
  - ① The branch predictor will predict not taken for branch A — very likely
  - ② The cache may contain the content of `array2[array1[16] * 512];` — possibly
  - ③ temp can potentially become the value of `array2[array1[16] * 512];` — where the security issues come from
  - ④ The program will raise an exception — impossible — not really, as x < array1_size

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

```
unsigned int array1_size = 16;

uint8_t array1[160] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 260};
uint8_t array2[256 * 512];

void bar(size_t x) {
  if (x < array1_size) { // Branch A: Taken if the statement is not going to be executed.
    temp &= array2[array1[x] * 512];
  }
}

void foo(size_t x) {
    int i = 0, j=0;
    for(j=0;j<10000;j++)
            bar(rand()%17);
}
```



# Illusion of a multi-core processor

- To create an illusion of a multi-core processor and allow the core to run instructions from multiple threads concurrently, how many of the following units in the processor must be duplicated?
  - ① Program counter
  - ② Register file
  - ③ ALUs
  - ④ Data cache
  - ⑤ Reorder buffer

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

- How many of the following about SMT are correct?
  - ① SMT makes processors with deep pipelines more tolerable to mis-predicted branches
  - ② SMT can ~~improve~~ the throughput of a single-threaded application
    hurt, b/c you are sharing resource with other threads.
  - ③ SMT processors can better utilize hardware during cache misses comparing with superscalar processors with the same issue width
  - ④ SMT processors can have higher cache miss rates comparing with superscalar processors with the same cache sizes when executing the same set of applications.

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