Midterm Review

Introduction to Computer Systems, Fall 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka Daniel Gearhardt
Ash Fujiyama Emily Shen

TAs:

Ahmed Abdellah
Angie Cao
August Fu
Caroline Begg
Cathy Cao
Claire Lu
Eric Sungwon Lee
Ethan Weisberg
Garrett O'Malley Kirsch
Hassan Rizwan
Iain Li
Jerry Wang
Juan Lopez
Keith Mathe
Maya Huizar
Meghana Vasireddy
Perrie Quek
Sidharth Roy
Sydnie-Shea Cohen
Vivi Li
Yousef AlRabiah

Midterm Review: Which Topic Next?

- Binary & 2C
- Binary C Programming
- C: Memory Diagrams
- C Programming: Strings & Output Parameters
- C Programming: Malloc & Double Pointers
- CMOS, PLA, Gates
- Gate Delay
- Combinatorial Logic: Mux

Logistics

- Midterm Exam: **This Thursday "in lecture"**
  - Details released on the course website
- Midterm Review in recitation
  - @4pm tomorrow (DRL 3C6)
  - Extra Recitation offering at 7:30 tomorrow (Towne 100)
- HW04 Sample solutions and grades posted yesterday
- HW05 Sample Solutions (and grades probably) posted tonight

Binary & 2C

- There are about 236 students in the class and 27 staff. If we wanted to assign each of these individuals a unique numerical ID, how many bits would each ID need to be?

- Translate:
  - -1 into 4-bit 2C
  - 7 into 4-bit 2c
  - 7 into 8-bit 2c
  - 5 into 3bit unsigned

Binary & 2C

- There are about 195 students in the class and 22 staff. If we wanted to assign each of these individuals a unique numerical ID, how many bits would each ID need to be?
  - 236 + 27 = 263. smallest power of 2 that is greater than 263 is 512 2^9. So 9 bits needed. (N bits can represent 2^N different values)
- Translate:
  - -1 into 4-bit 2C
    - 1111 // -8 + 4 + 2 + 1 = -1 (and -1 is always the all 1 bit pattern)
  - 7 into 4-bit 2c
    - 0111 // $(-1 * 0 * 2^3) + 1 * 2^2 + 1 * 2^1 + 1*2^0 = 0 + 4 + 2 + 1 = 7$
  - 7 into 8-bit 2c:
    - 0000 0111 already, had 7 from above with 4 bits. Add leading zeros to leave the numerical value unchanged.
  - 5 into 3bit unsigned 101 // $1 * 2^2 + 0 * 2^1 + 1*2^0 = 4 + 0 + 1 = 5$

Binary C Programming

- Write the function `reverse_bits()` which takes an unsigned integer and returns a new unsigned integer but with the bits reversed
  - Assume `unsigned int` is 32 bits long
  - Input: `0000 … 0001` returns `1000 … 0000` (only 1 bit is a 1)
  - Input : `1111 … 1111` returns `1111 … 1111` (all bits are 1)

```
unsigned int reverse_bits(unsigned int num) {









}
```

Binary C Programming

- Write the function `reverse_bits()` which takes an unsigned integer and returns a new unsigned integer but with the bits reversed
  - Assume `unsigned int` is 32 bits long

```c
unsigned int reverse_bits(unsigned int num) {
  unsigned int result = 0;
  for (int i = 0; i < 32 ; i++) {
    result = result << 1;  // shift result to make room
                           // for next bit

    unsigned int lsb = num & 1; // get lsb of num
    result = result | lsb;  // add the lsb to result

    num = num >> 1;  // shift num to right to get next bit
  }
  return result;
}
```

C: Memory Diagrams

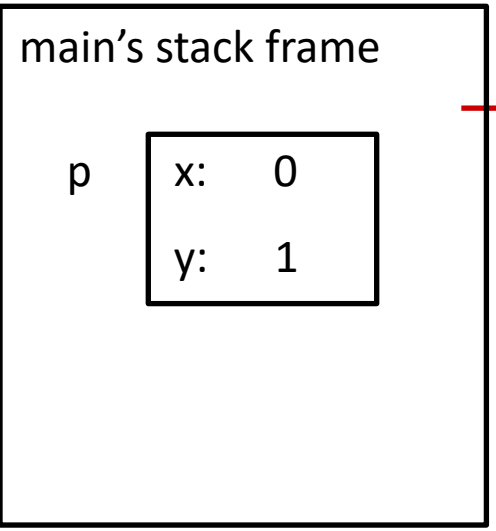
```c
typedef struct {
  int x;
  int y;
} pair;

typedef struct {
  pair* data;
  size_t len;
} pair_arr;
```

```c
pair_arr make_pair_arr(size_t len, pair p) {
  pair_arr result;
  result.len = len;
  result.data = malloc(sizeof(pair) * len);
  for (size_t i = 0; i < len; i++) {
    p.x += i;
    p.y += i;
    result.data[i] = p;
  }
  return result;
}
```

```c
int main() {
  pair p = (pair) {0, 1};
  pair_arr a = make_pair_arr(3, p);
  printf("%d %d\n", p.x, p.y);
  for (size_t i = 0; i < a.len; i++) {
    p = a.data[i];
    printf("%d %d\n", p.x, p.y);
  }
}
```

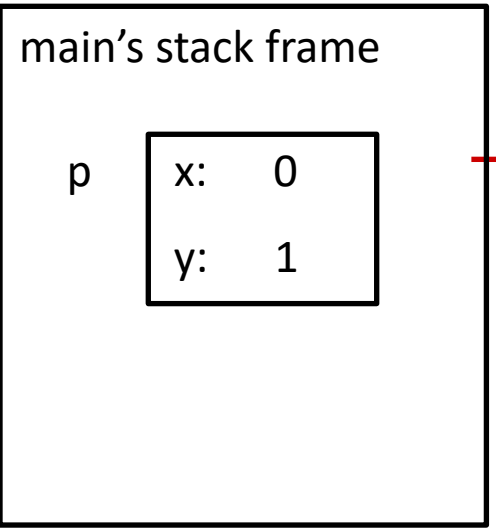
What does this print?
What memory errors are there?
How do we fix them?

C: Memory Diagrams




```c
int main() {
  pair p = (pair) {0, 1};
  pair_arr a = make_pair_arr(3, p);
  printf("%d %d\n", p.x, p.y);
  for (size_t i = 0; i < a.len; i++) {
    p = a.data[i];
    printf("%d %d\n", p.x, p.y);
  }
}
```

C: Memory Diagrams




```c
int main() {
  pair p = (pair) {0, 1};
  pair_arr a = make_pair_arr(3, p);
  printf("%d %d\n", p.x, p.y);
  for (size_t i = 0; i < a.len; i++) {
    p = a.data[i];
    printf("%d %d\n", p.x, p.y);
  }
}
```

C: Memory Diagrams

main's stack frame

p

x:	0
y:	1

m_p_a's stack frame

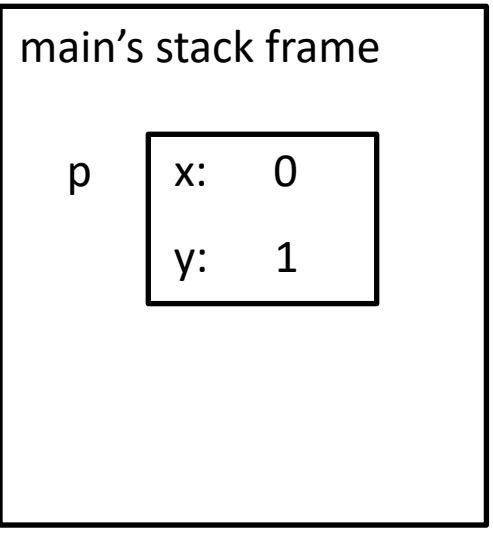
p

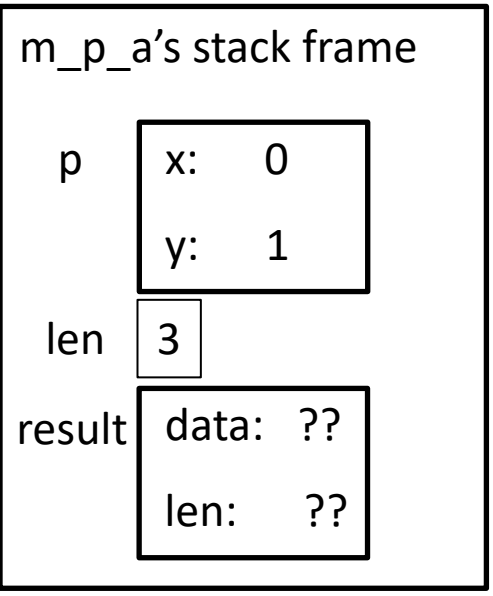
x:	0
y:	1

len | 3 |

```
pair_arr make_pair_arr(size_t len, pair p) {
  pair_arr result;
  result.len = len;
  result.data = malloc(sizeof(pair) * len);
  for (size_t i = 0; i < len; i++) {
    p.x += i;
    p.y += i;
    result.data[i] = p;
  }
  return result;
}
```

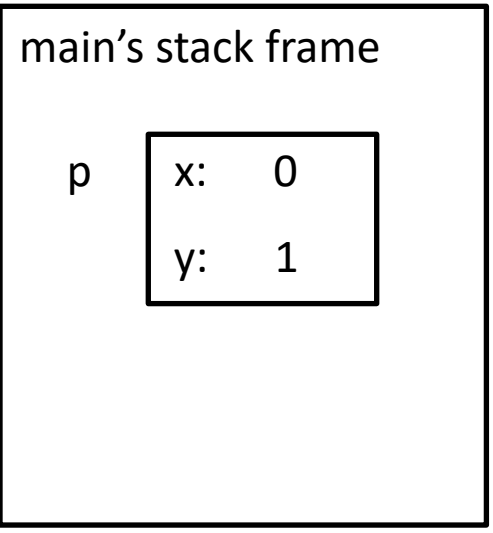
C: Memory Diagrams

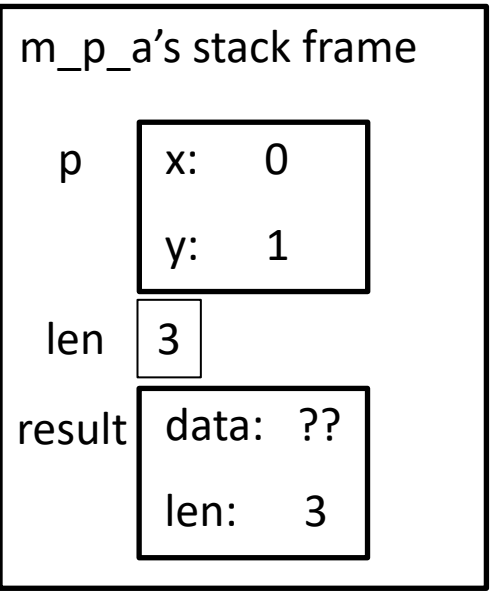





```c
pair_arr make_pair_arr(size_t len, pair p) {
  pair_arr result;
  result.len = len;
  result.data = malloc(sizeof(pair) * len);
  for (size_t i = 0; i < len; i++) {
    p.x += i;
    p.y += i;
    result.data[i] = p;
  }
  return result;
}
```

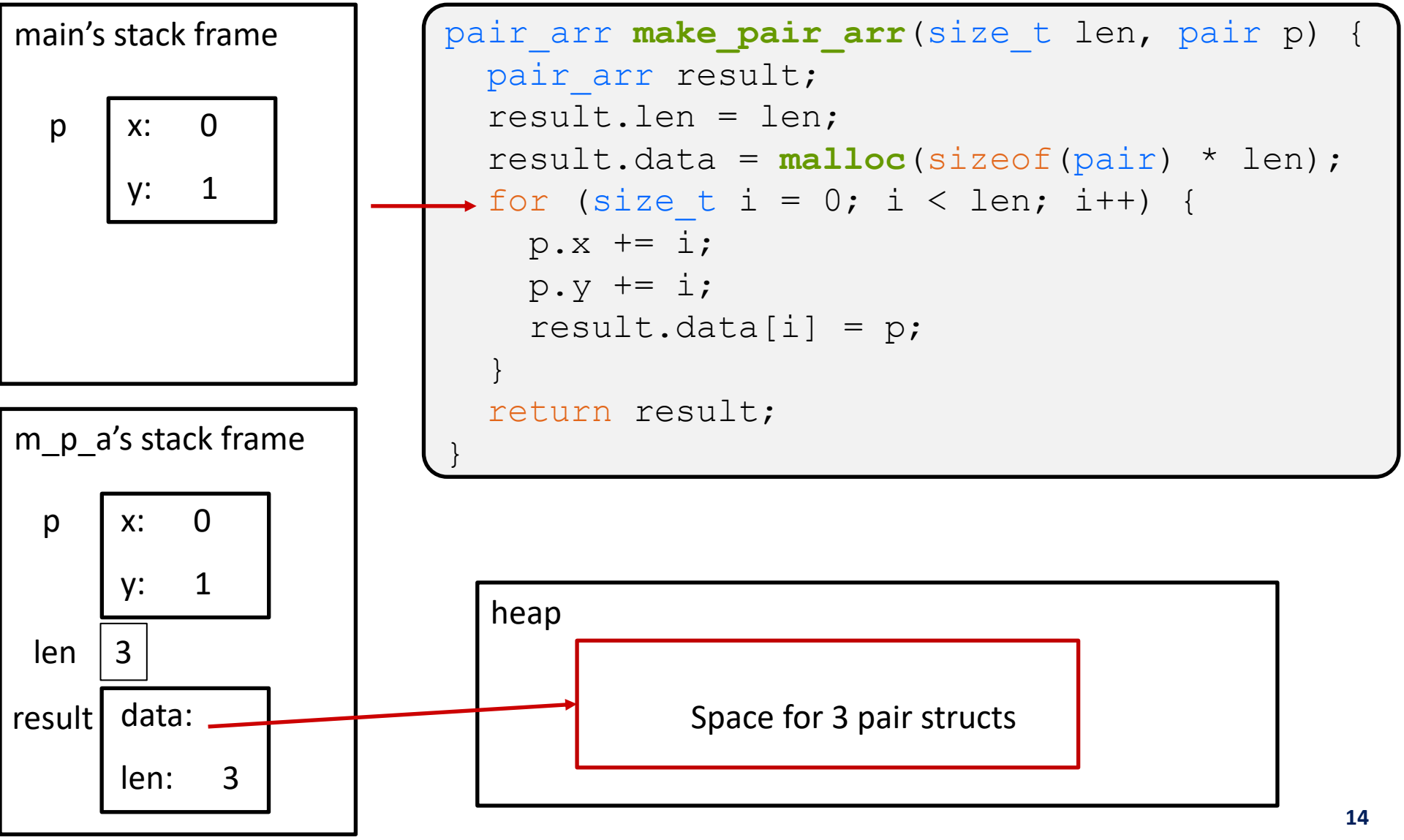
C: Memory Diagrams





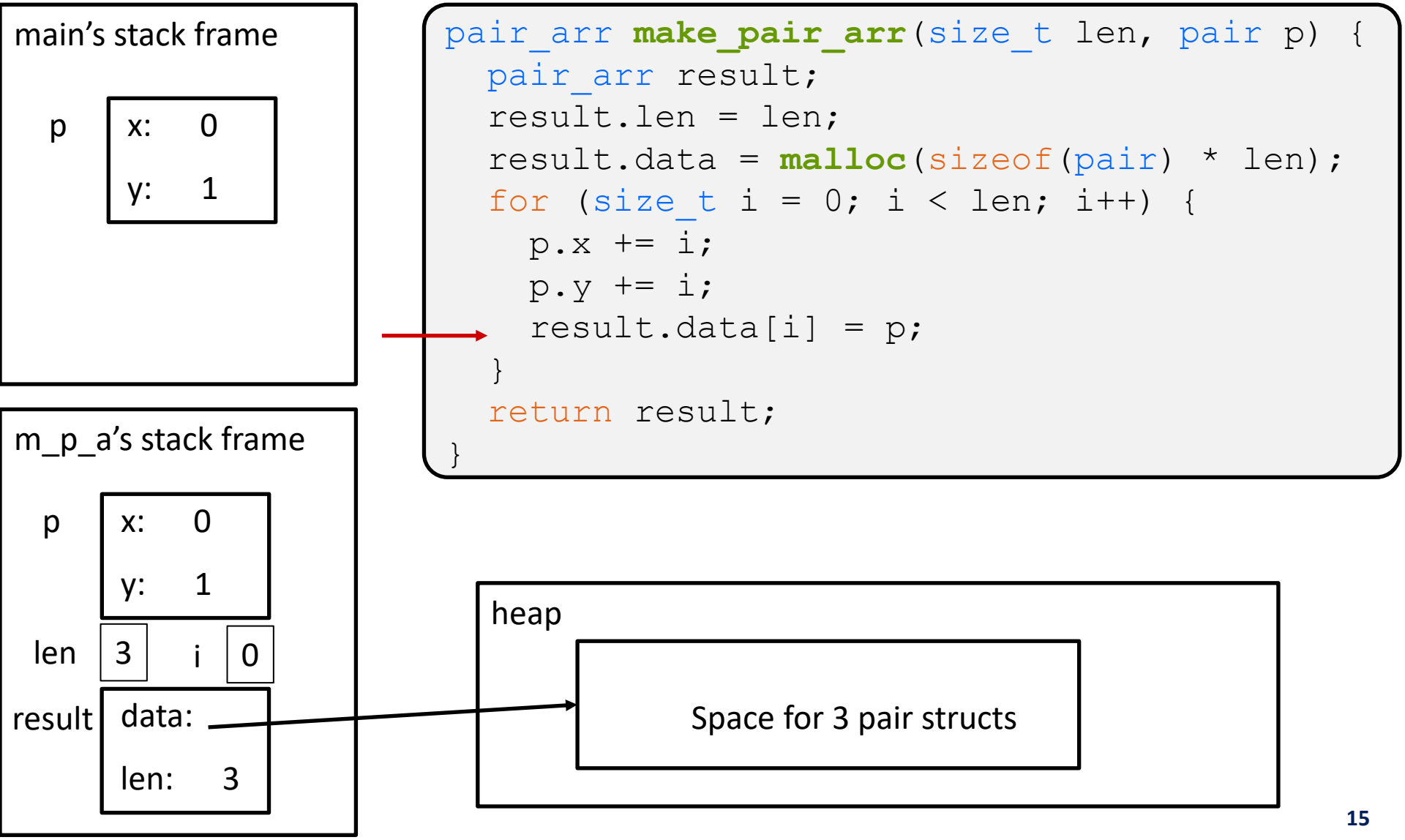

```
pair_arr make_pair_arr(size_t len, pair p) {
  pair_arr result;
  result.len = len;
  result.data = malloc(sizeof(pair) * len);
  for (size_t i = 0; i < len; i++) {
    p.x += i;
    p.y += i;
    result.data[i] = p;
  }
  return result;
}
```

C: Memory Diagrams



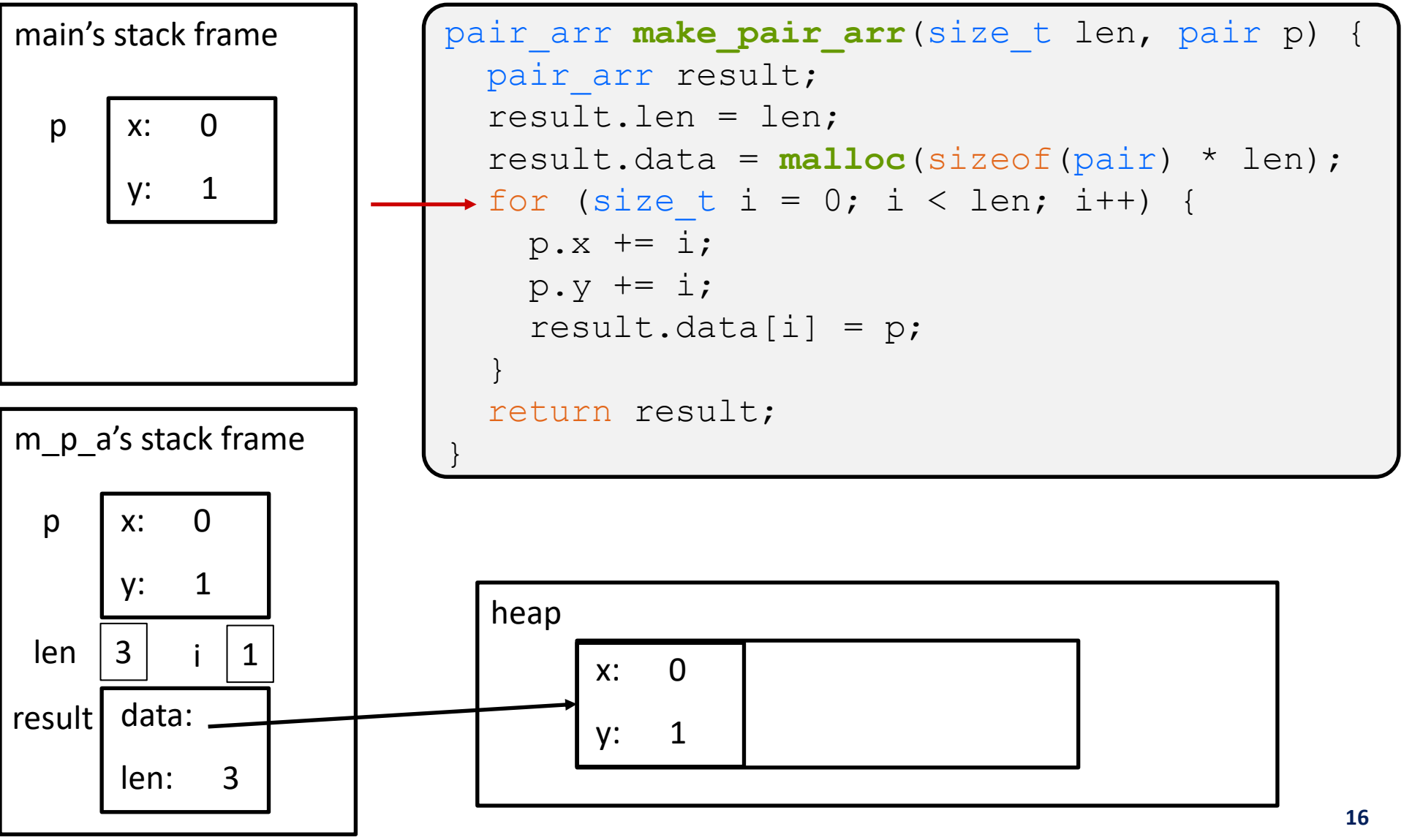

```
pair_arr make_pair_arr(size_t len, pair p) {
  pair_arr result;
  result.len = len;
  result.data = malloc(sizeof(pair) * len);
  for (size_t i = 0; i < len; i++) {
    p.x += i;
    p.y += i;
    result.data[i] = p;
  }
  return result;
}
```

C: Memory Diagrams

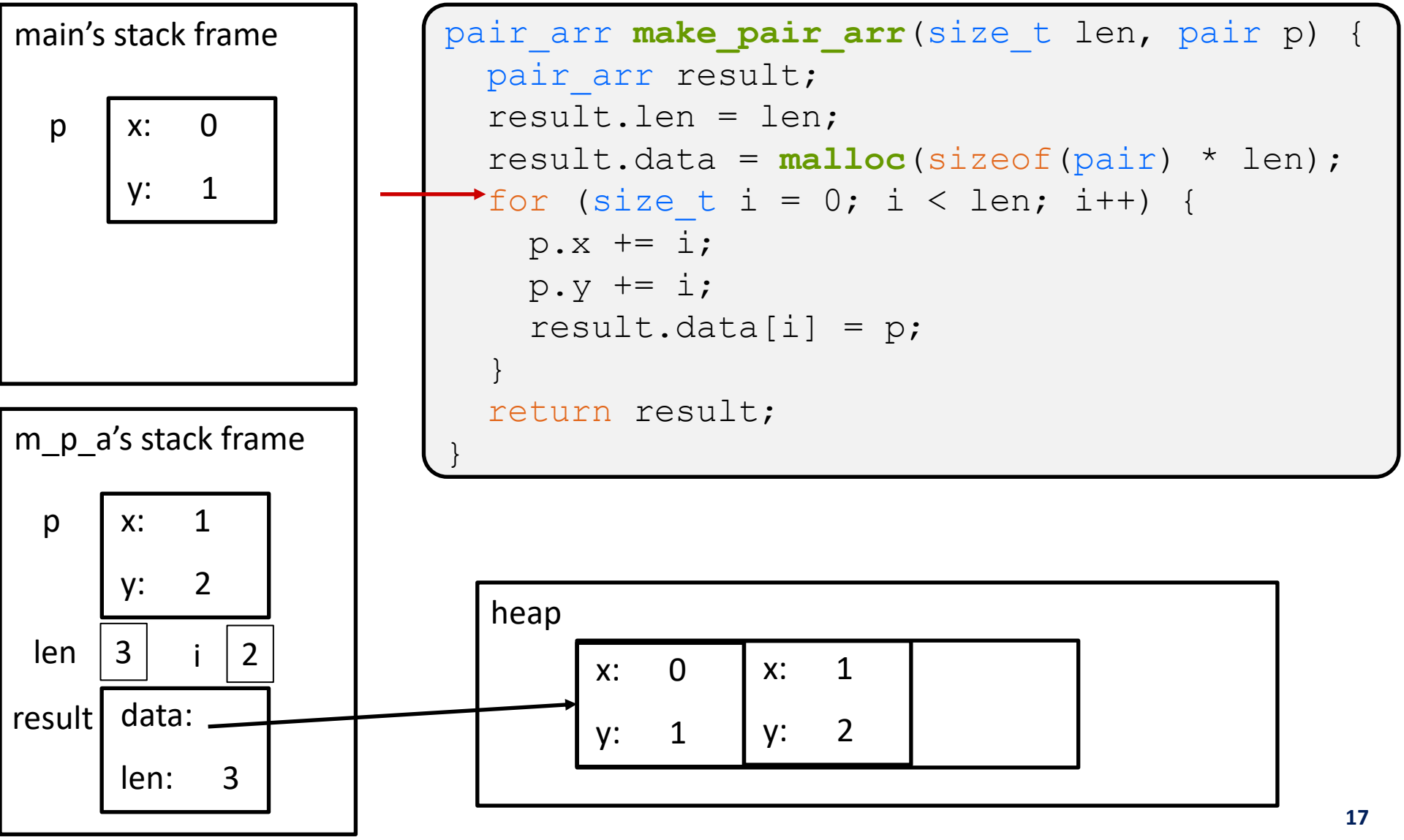



```
pair_arr make_pair_arr(size_t len, pair p) {
  pair_arr result;
  result.len = len;
  result.data = malloc(sizeof(pair) * len);
  for (size_t i = 0; i < len; i++) {
    p.x += i;
    p.y += i;
    result.data[i] = p;
  }
  return result;
}
```

C: Memory Diagrams

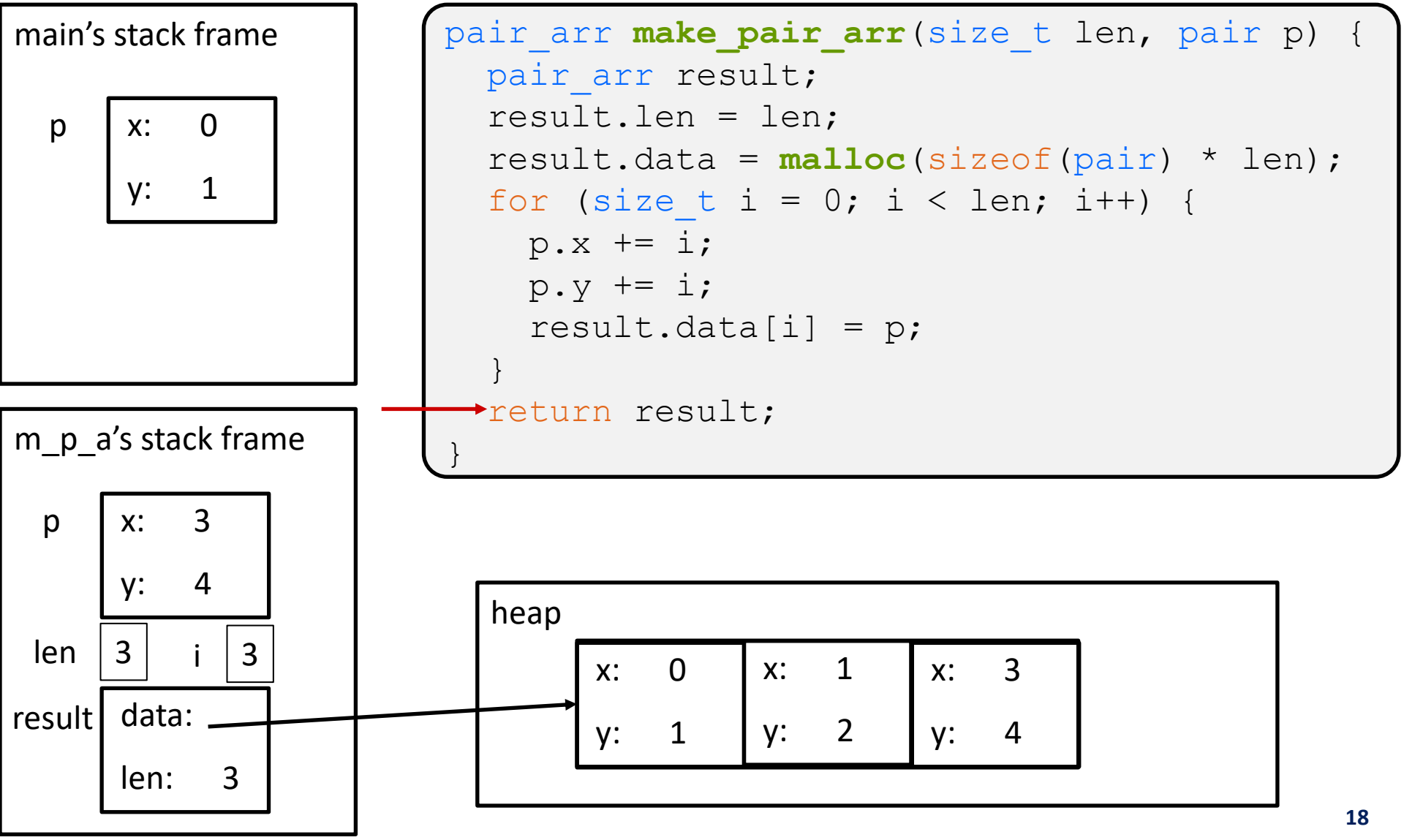


C: Memory Diagrams



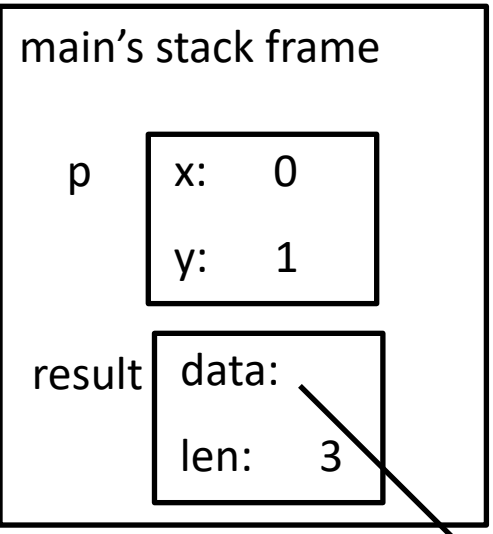

```
pair_arr make_pair_arr(size_t len, pair p) {
  pair_arr result;
  result.len = len;
  result.data = malloc(sizeof(pair) * len);
  for (size_t i = 0; i < len; i++) {
    p.x += i;
    p.y += i;
    result.data[i] = p;
  }
  return result;
}
```

C: Memory Diagrams

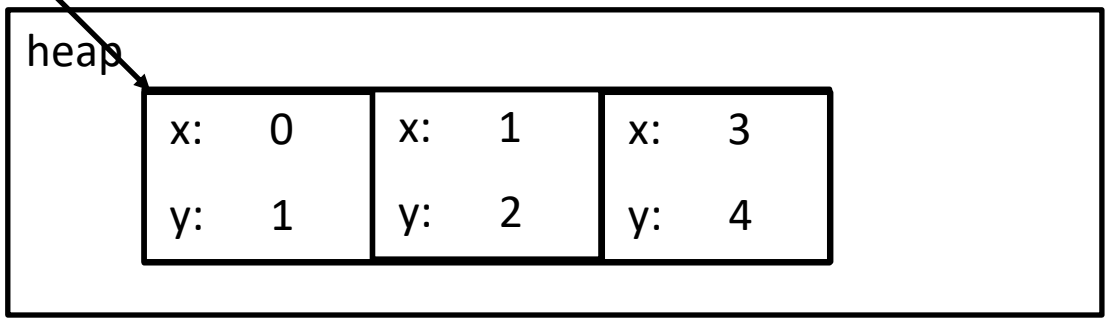



```c
pair_arr make_pair_arr(size_t len, pair p) {
  pair_arr result;
  result.len = len;
  result.data = malloc(sizeof(pair) * len);
  for (size_t i = 0; i < len; i++) {
    p.x += i;
    p.y += i;
    result.data[i] = p;
  }
  return result;
}
```

C: Memory Diagrams




```c
int main() {
  pair p = (pair) {0, 1};
  pair_arr a = make_pair_arr(3, p);
  printf("%d %d\n", p.x, p.y);
  for (size_t i = 0; i < a.len; i++) {
    p = a.data[i];
    printf("%d %d\n", p.x, p.y);
  }
}
```



C Strings & Output Params

- Complete the following function (on Codio)

```
// given a string, allocates and creates a new duplicate
// of it and returns it through the output parameter "out".
// Returns false on error, returns true otherwise

bool str_duplicate(char* str, char** out) {









}
```

C Strings & Output Params

NOTE: There are other possible solutions

- Complete the following function

```
// given a string, allocates and creates a new duplicate
// of it and returns it through the output parameter "out".
// Returns false on error, returns true otherwise

bool str_duplicate(char* str, char** out) {
  char* res = malloc((strlen(str) + 1) * sizeof(char));
  *out = res;
  while(*str) {
    *res = *str
    str++;
    res++;
  }
  *res = *str;  // null terminator
  return true;
}
```

C Strings & Output Params

- Complete the main function

```
// given a string, duplicates it and returns it through
// the output parameter "out". Returns false on error
// returns true otherwise
bool str_duplicate(char* str, char** out);

// duplicates a string literal,
// prints the duplicate, and runs without errors
int main(int argc, char** argv) {
  char* sample = "Hello World!";




}
```

C Strings & Output Params

NOTE: There are other possible solutions

- Complete the main function

```c
// given a string, duplicates it and returns it through
// the output parameter "out". Returns false on error
// returns true otherwise
bool str_duplicate(char* str, char** out);

// duplicates a string literal,
// prints the duplicate, and runs without errors
int main(int argc, char** argv) {
  char* sample = "Hello World!";

  char* dup;
  str_duplicate(sample, &dup);
  printf("%s", dup);
  free(dup);
  return EXIT_SUCCESS;
}
```

This problem may be on the harder side.

If space would have given you a memory diagram of the output

C Programming: Malloc & Double Pointers

- We want to make a module that implements 2d matrices in C. We define the following struct which holds a dynamically allocated 2-dimensional array.

```
typedef struct {
  int** data;
  int rows;
  int cols;
} matrix;
```

- Implement the `create_matrix` function which creates a matrix on the heap with the specified rows and cols. Assume malloc does not fail. Zero out all data in the matrix.
- Example: create_matrix(2, 3) should create a 2x3 matrix. data points to 2 `int*`, each of those point to 3 `int`s.

```
matrix* create_matrix(int rows, int cols) {
  // Implement this function. You need more than 1 line.
}
```

This problem may be on the harder side.

If space would have given you a memory diagram of the output

C Programming: Malloc & Double Pointers

```
matrix* create_matrix(int rows, int cols) {
  matrix *m  = malloc(sizeof(matrix));
  m->rows = rows;
  m->cols = cols;

  m->data = malloc(rows * sizeof(int*));

  for (int i = 0; i < rows; i++) {
    m->data[i] = malloc(cols * sizeof(int));
    for (int j = 0; j < cols; j++) {
      m->data[i][j] = 0;
    }
  }
  return m;
}
```

C Programming: Malloc & Double Pointers

- We want to make a module that implements 2d matrices in C. We define the following struct which holds a dynamically allocated 2-dimensional array.

```
typedef struct {
  int** data;
  int rows;
  int cols;
} matrix;
```

- Implement the `free_matrix()` function which deallocates the matrix allocated in `create_matrix()`
- Example: create_matrix(2, 3) should create a 2x3 matrix. data points to 2 `int*`, each of those point to 3 `int`s.

```
void free_matrix(matrix* m) {
  // Implement this function.



}
```

C Programming: Malloc & Double Pointers

```c
matrix* create_matrix(int rows, int cols) {
  matrix *m  = malloc(sizeof(matrix));
  m->rows = rows;
  m->cols = cols;

  m->data = malloc(rows * sizeof(int*));

  for (int i = 0; i < rows; i++) {
    m->data[i] = malloc(cols * sizeof(int));
    for (int j = 0; j < cols; j++) {
      m->data[i][j] = 0;
    }
  }
  return m;
}
```

```c
void free_matrix(matrix* m) {
  // Implement this function.


}
```

C Programming: Malloc & Double Pointers

```
void free_matrix(matrix* m) {
  for (int i = 0; i < m->rows; i++) {
    free(m->data[i]);
  }
  free(m->data);
  free(m);
}
```

CMOS, PLAS, GATES

- Create a circuit that takes in an unsigned 4-bit input I ($I_3I_2I_1I_0$), and outputs a 1 if and only if the 4-bit input is a non-zero multiple of 7
  - List the outputs that result in a 1 for the output
  - Create a corresponding CMOS circuit
    - Can assume you have the inverses of the Input bits
  - Create a corresponding PLA circuit
  - Create a corresponding gate level non-PLA circuit

CMOS, PLAS, GATES

- Create a circuit that takes in an unsigned 4-bit input I ($I_3I_2I_1I_0$), and outputs a 1 if and only if the 4-bit input is a non-zero multiple of 7
  - List the outputs that result in a 1 for the output
  - 7 (0b0111) and 14 (0b1110)

CMOS

- Create a circuit that takes in an unsigned 4-bit input I ($I_3I_2I_1I_0$), and outputs a 1 if and only if the 4-bit input is a non-zero multiple of 7. You can assume you have inverse of the input signals.
  - Overall Expression: $(\sim I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, I_0) \mid (I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, \sim I_0)$

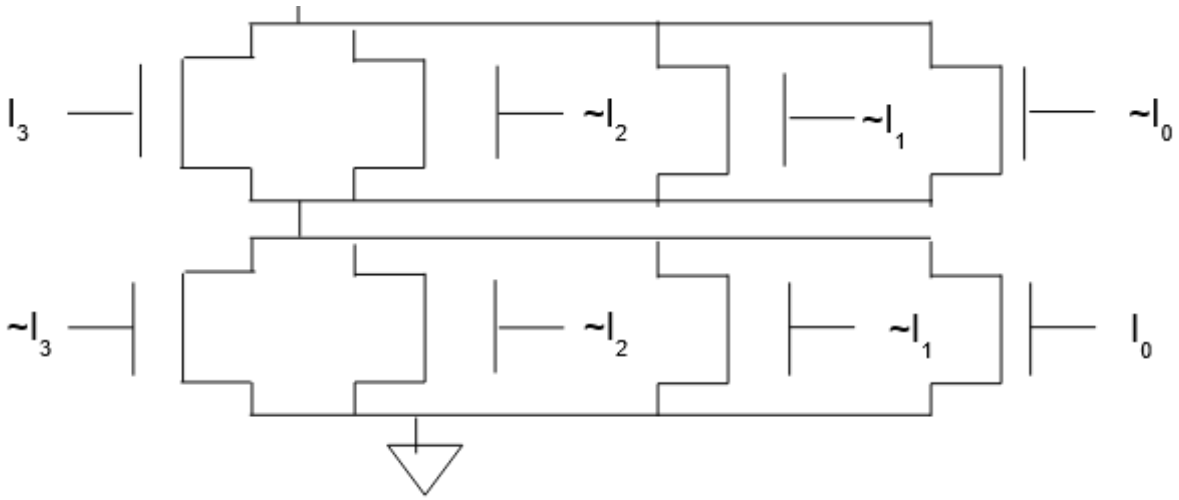
CMOS Strategy 1 (Starting with PDN)

- Overall Expression: $(\sim I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, I_0) \mid (I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, \sim I_0)$
- PDN Expression:
  - $\sim((\sim I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, I_0) \mid (I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, \sim I_0))$
  - $\sim(\sim I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, I_0) \,\&\, \sim(I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, \sim I_0)$
  - $(I_3 \mid \sim I_2 \mid \sim I_1 \mid \sim I_0) \,\&\, (\sim I_3 \mid \sim I_2 \sim I_1 \mid I_0)$

CMOS Strategy 1 (Starting with PDN)

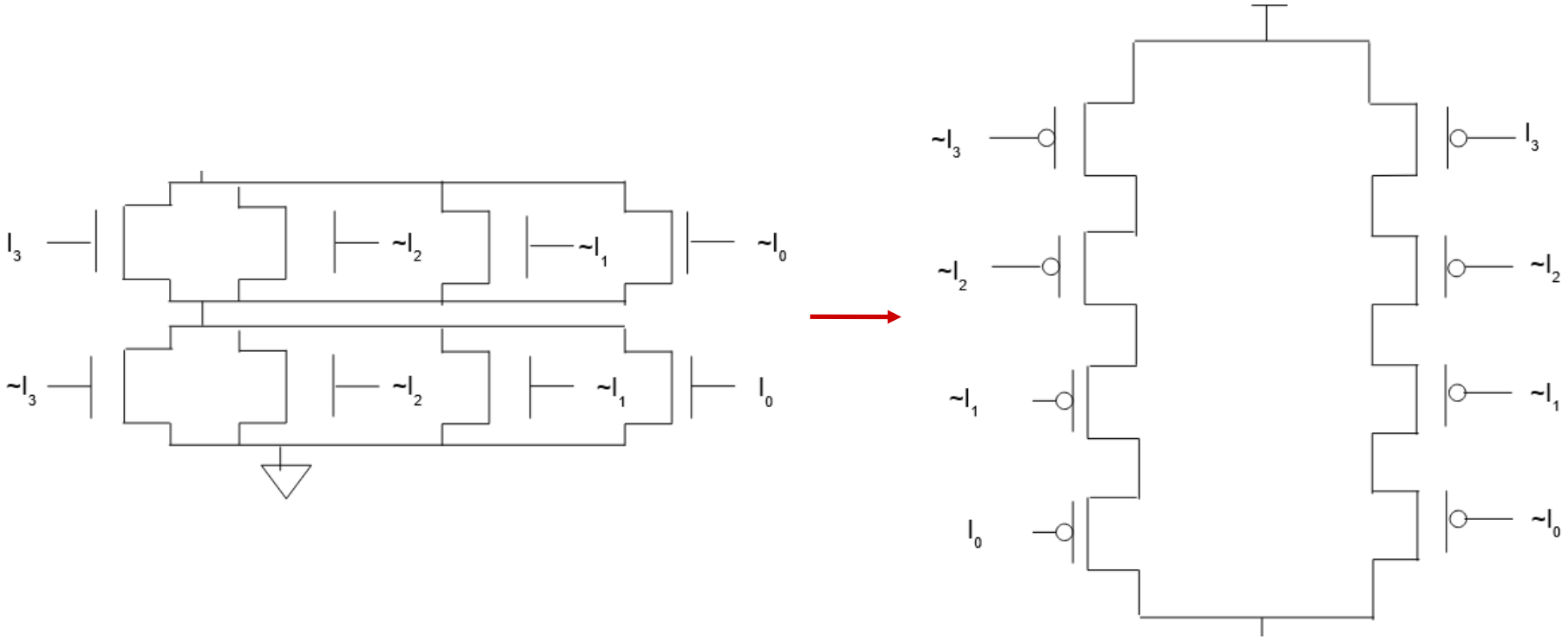
- Overall Expression: $(\sim I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, I_0) \mid (I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, \sim I_0)$
- PDN Expression:
  - $\sim((\sim I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, I_0) \mid (I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, \sim I_0))$ // negate
  - $\sim(\sim I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, I_0) \,\&\, \sim(I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, \sim I_0)$ // De Morgan's
  - $(I_3 \mid \sim I_2 \mid \sim I_1 \mid \sim I_0) \,\&\, (\sim I_3 \mid \sim I_2 \sim I_1 \mid I_0)$ // De Morgan's

- Translated to PDN:



CMOS Strategy 1 (Starting with PDN)

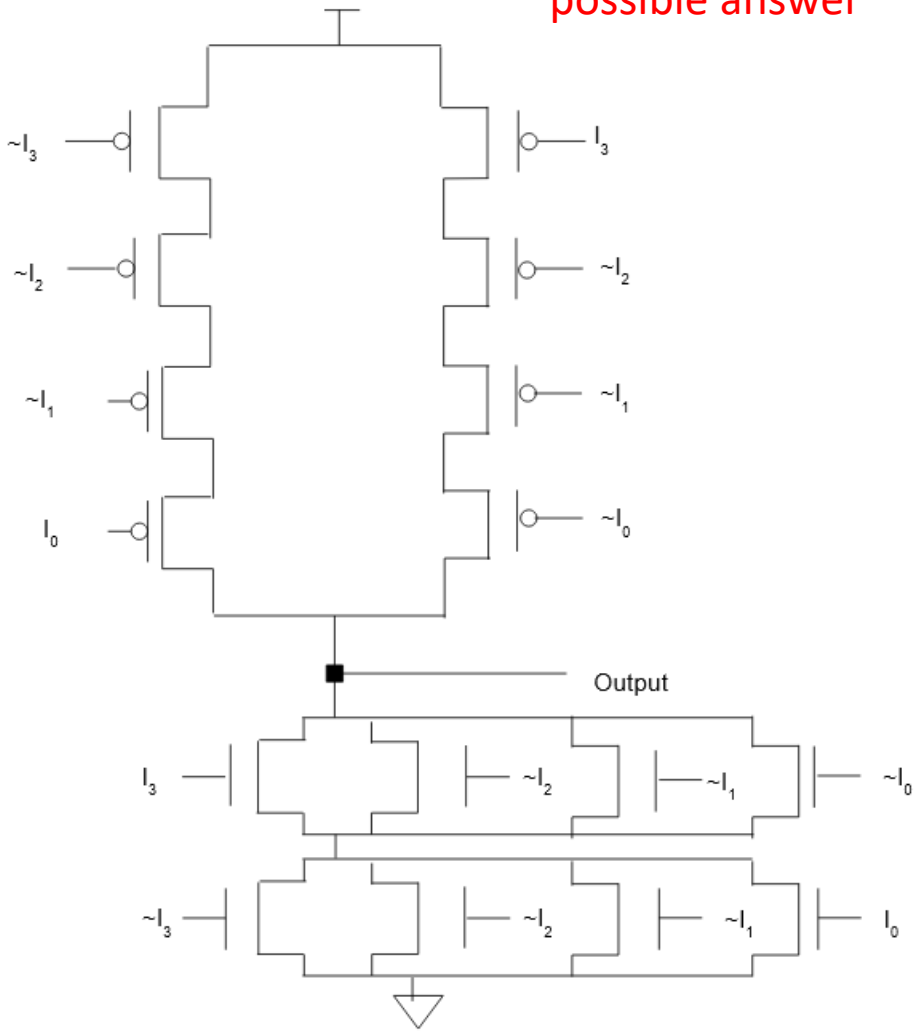
- Flip PDN into PUN:



CMOS Strategy 1 (Starting with PDN)

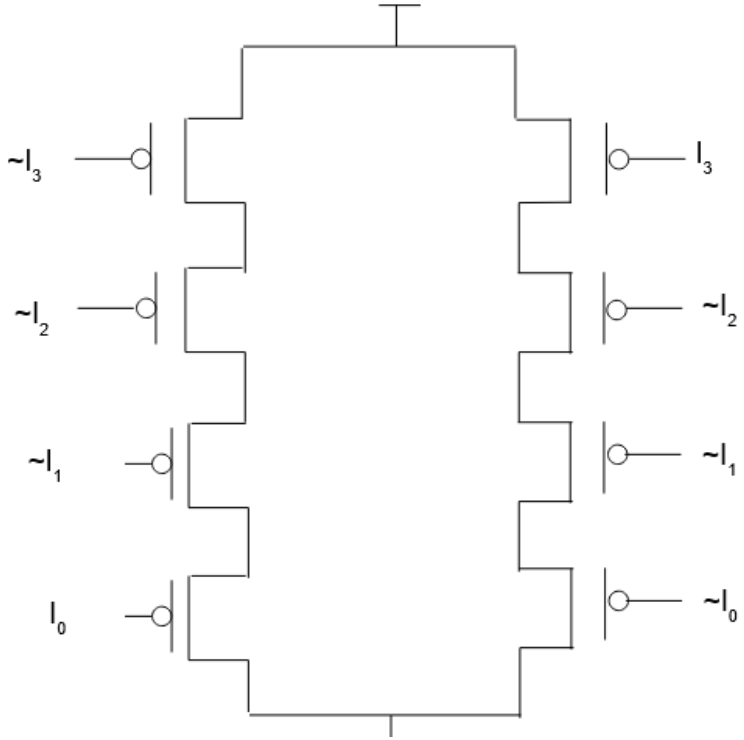
- Connect PDN and PUN:

Not the only possible answer



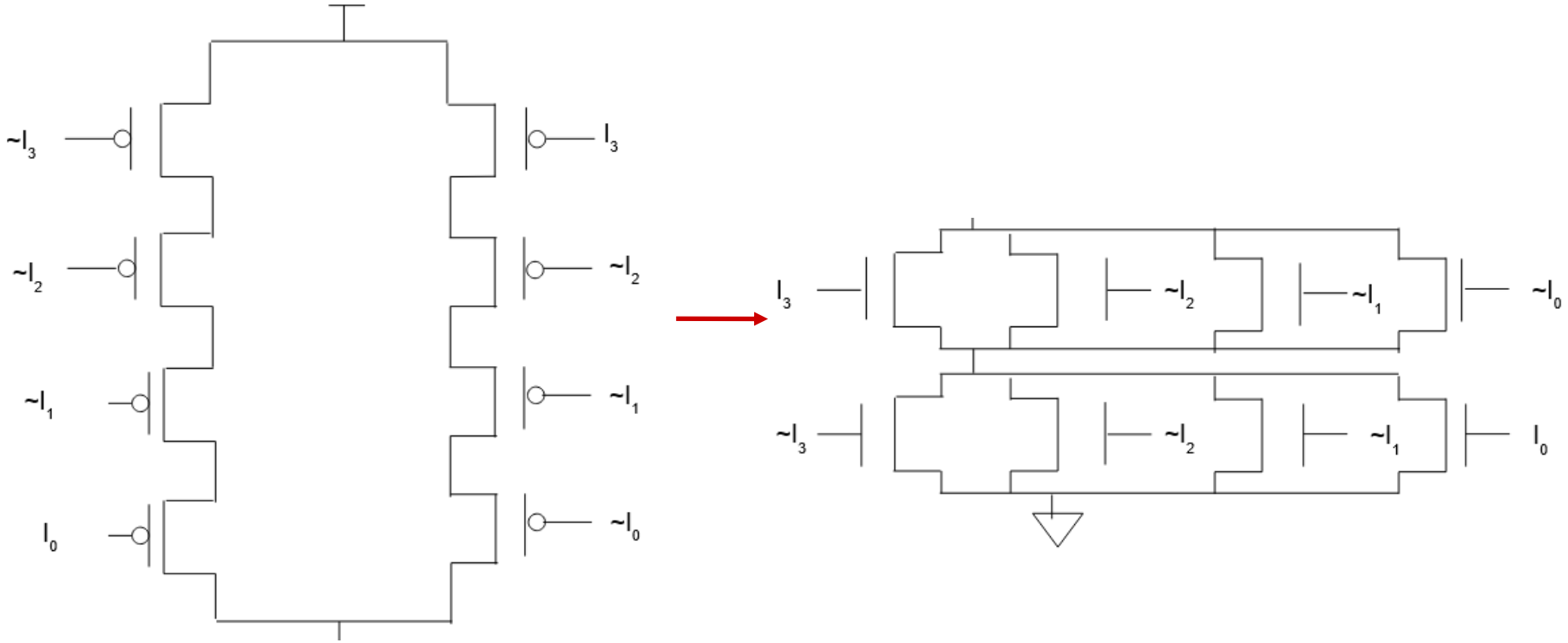
CMOS Strategy 2 (Starting with PUN)

- Take the original expression:
  - $(I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, \sim I_0) \mid (\sim I_3 \,\&\, I_2 \,\&\, I_1 \,\&\, I_0)$
- Translate it directly into PDN but add a negation to each input
  - This is because PMOS transistors are "naturally negating"
  - E.g., $\sim I_3$ becomes $\sim\sim I_3 == I_3$



CMOS Strategy 2 (Starting with PUN)

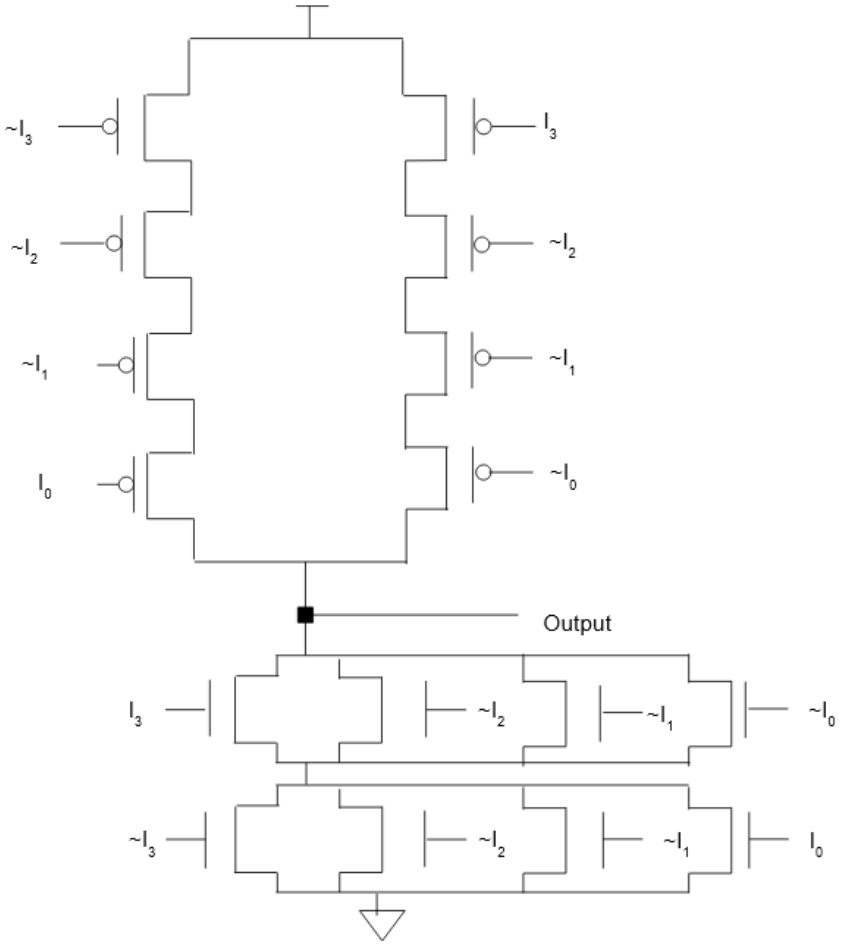
- Flip PUN to get PDN



CMOS Strategy 2 (Starting with PUN)

- Connect the PDN and PUN

Not the only possible answer

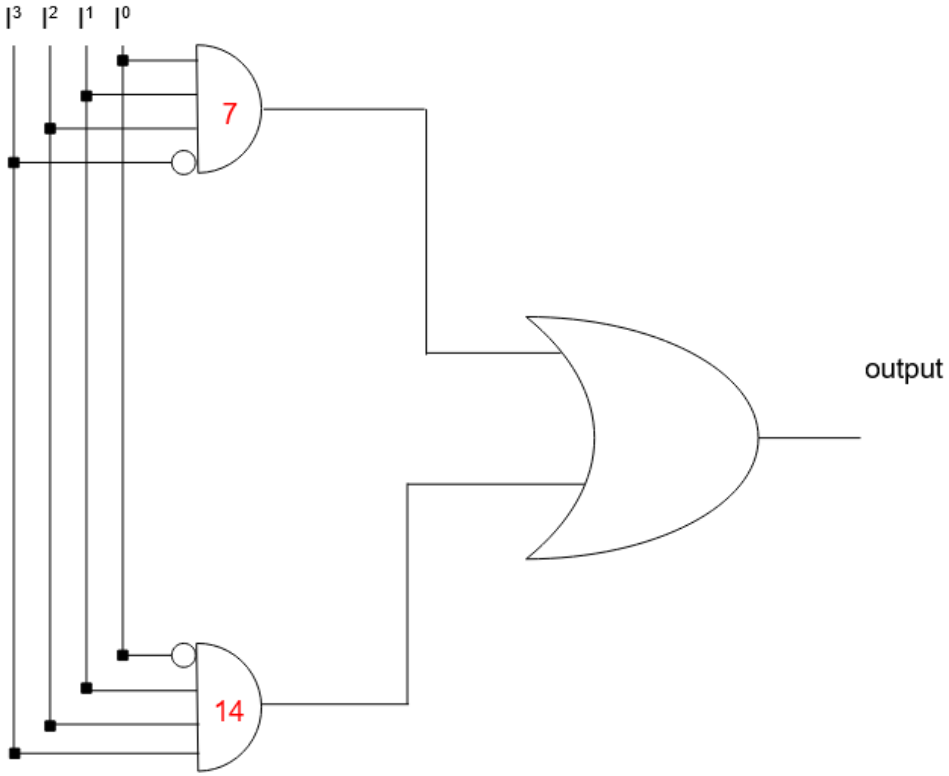


PLA

- Create a circuit that takes in an unsigned 4-bit input I ($I_3I_2I_1I_0$), and outputs a 1 if and only if the 4-bit input is a non-zero multiple of 7

PLA

- Create a circuit that takes in an unsigned 4-bit input I ($I_3I_2I_1I_0$), and outputs a 1 if and only if the 4-bit input is a non-zero multiple of 7



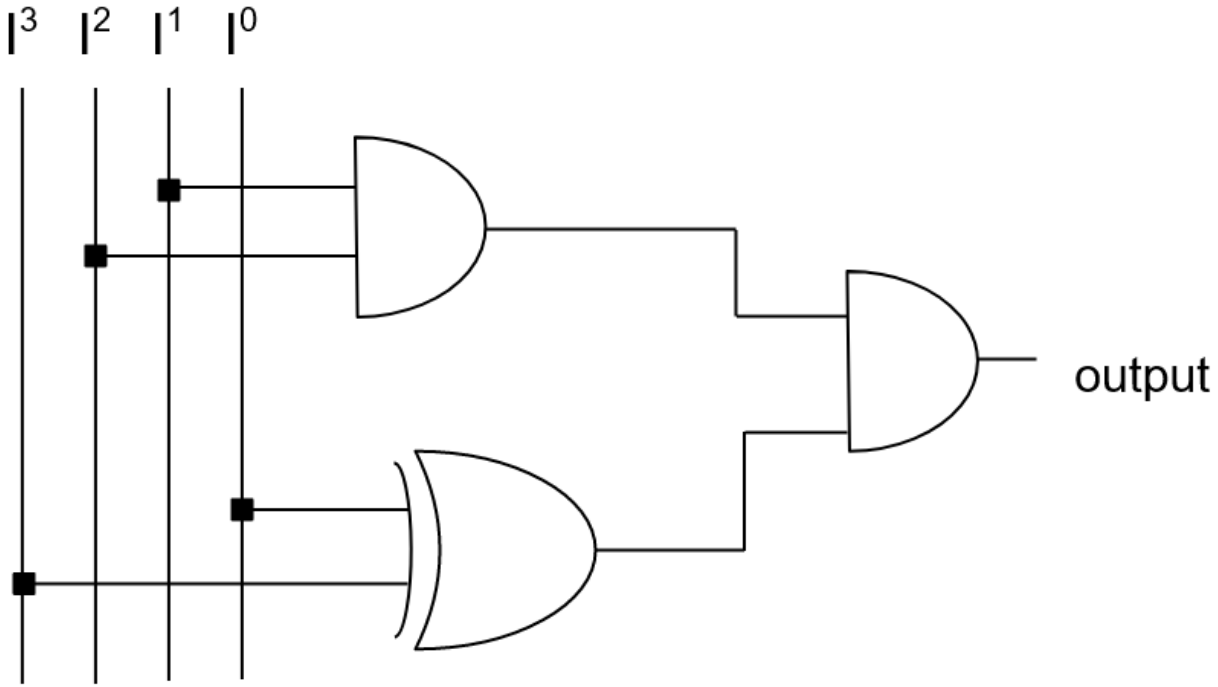
Non-PLA

- Create a circuit that takes in an unsigned 4-bit input I ($I_3I_2I_1I_0$), and outputs a 1 if and only if the 4-bit input is a non-zero multiple of 7

Non-PLA

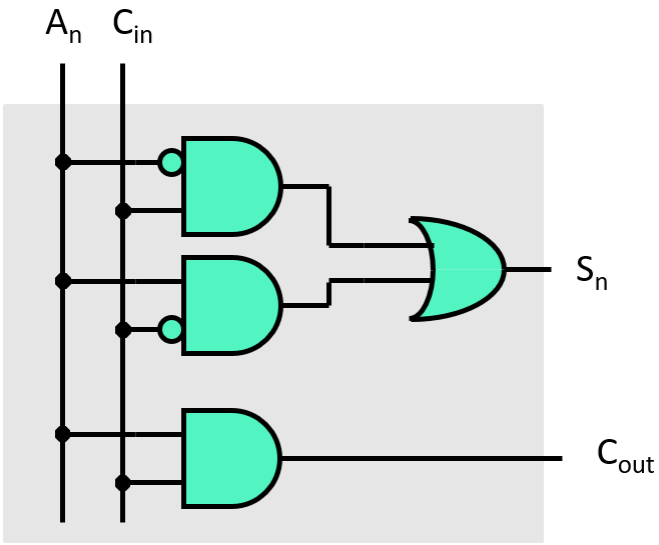
Not the only possible answer

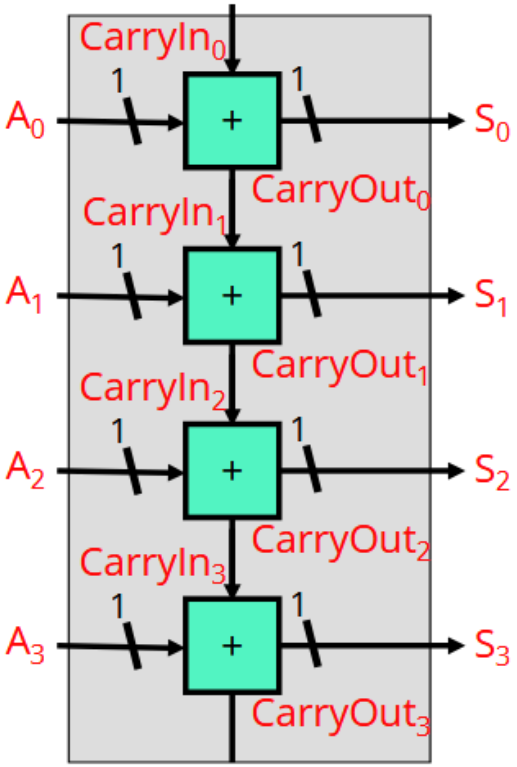
- Create a circuit that takes in an unsigned 4-bit input I ($I_3I_2I_1I_0$), and outputs a 1 if and only if the 4-bit input is a non-zero multiple of 7



Gate delay pt.1

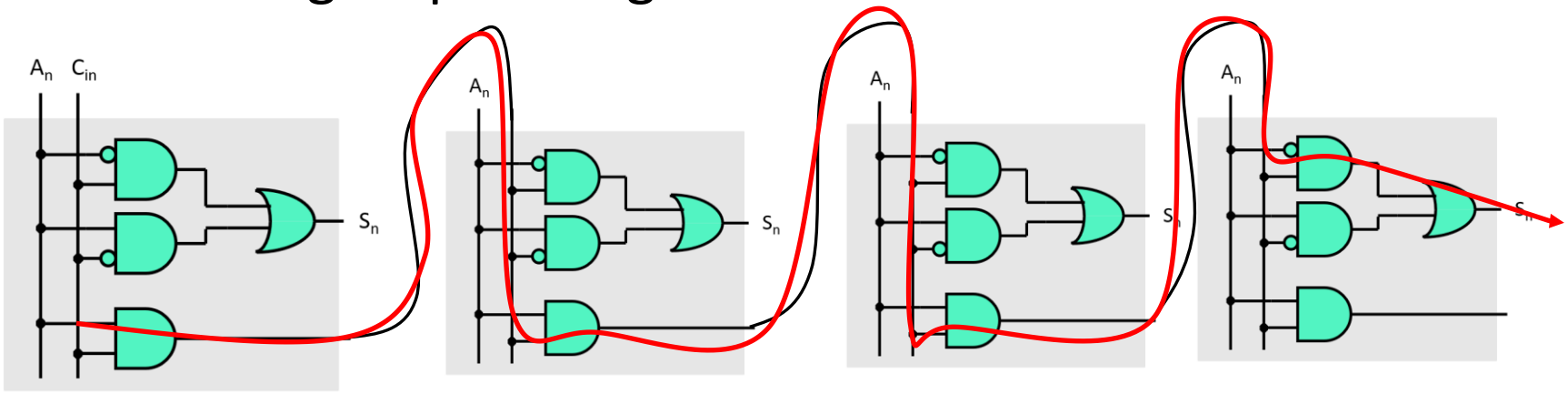
- Given the 4-bit incrementor that we created in lecture, how long do we have to wait to make sure that the output of the incrementor matches the input?
  - Assume that each gate has a 1ns delay
  - You can ignore delay from inverters





Gate delay pt.1 Answer

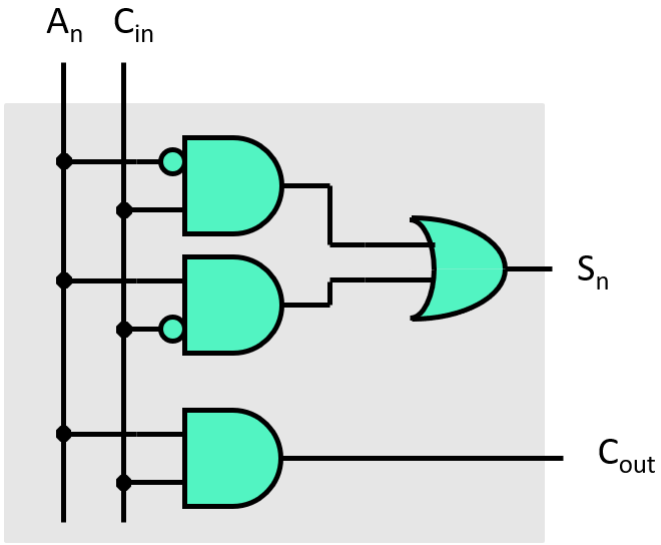
- Find the longest path of gates in the circuit

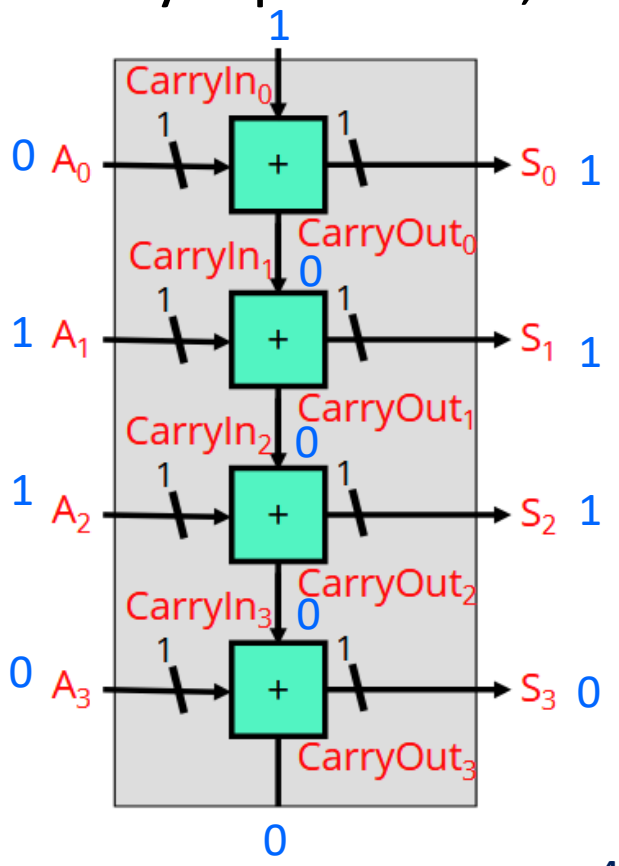


  - Longest path goes through 5 gates, so we must wait 5ns for the whole circuit to stabilize and produce the correct output

Gate delay pt.2

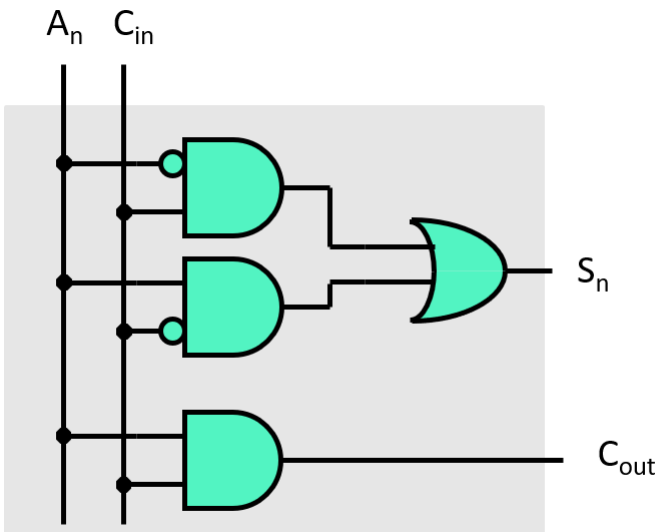
- The 4-bit incrementor that we created in lecture is currently in a stable state showing the output of 1 + 0110. If the A input signals were to simultaneously flip to 0111, what would all signals be after 2ns
  - Assume that each gate has a 1ns delay
  - Ignore delays from inverters
  - $CarryIn_n$ stays the same

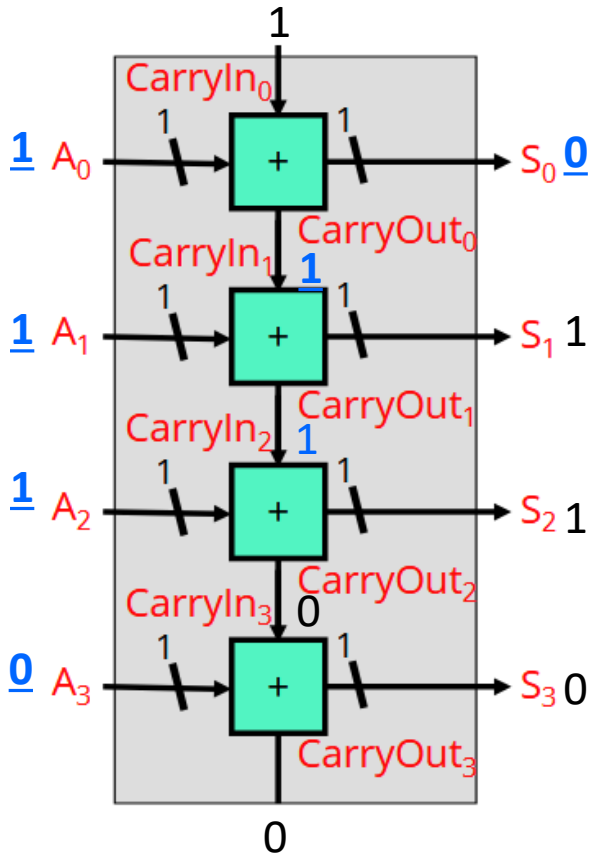




Gate delay pt.2

- To see what changes after 2ns, see which outputs have all paths that are 2 gate or less.
- Answer:
  - S_0 would update
  - $Cout_0$ would update
  - $Cout_1$ would update
  - S_1 would be partially computed, but not fully updated





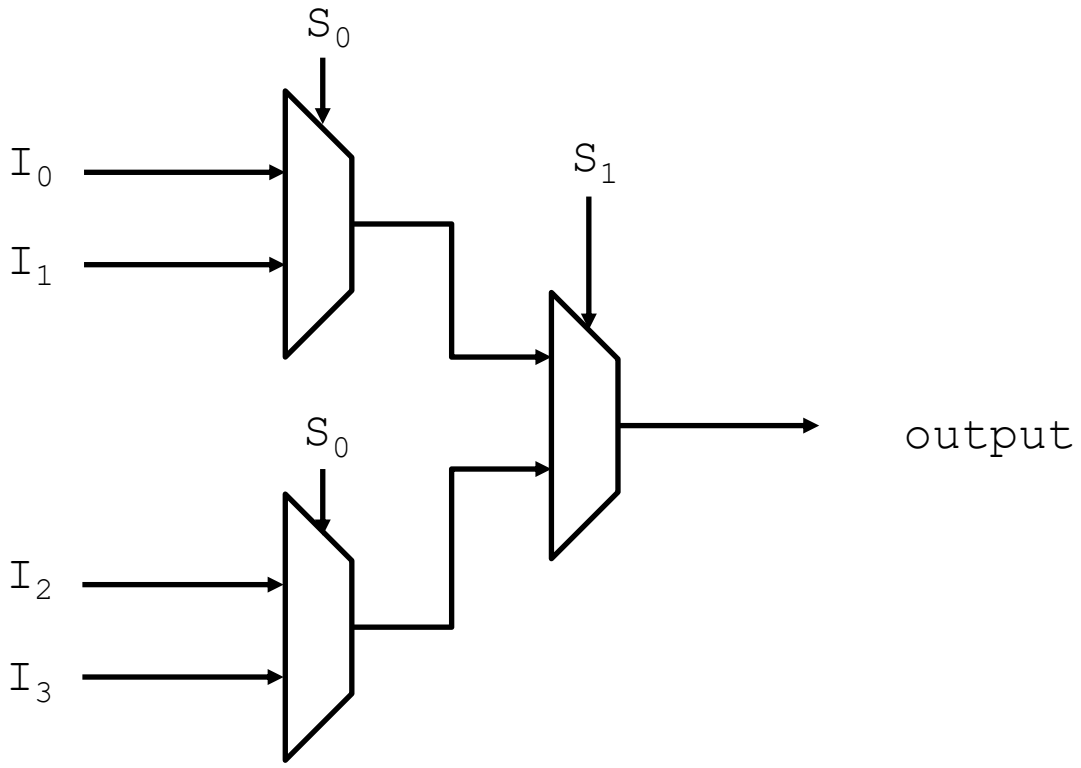
Combinatorial Logic: Mux

- Given inputs I_0 I_1 I_2 I_3 and selector bits S_1 and S_0 draw a logic circuit using 2-to-1 muxes to implement the 4-to-1 MUX. You can assume you have access to each bit/wire.

- Requirements:
  - If S_1S_0 == 00, the output should be I_0
  - If S_1S_0 == 01, the output should be I_1
  - If S_1S_0 == 10, the output should be I_2
  - If S_1S_0 == 11, the output should be I_3
  - Clearly label the inputs, select lines, and output
- Yes, you are allowed to use more than one 2-to-1 mux

Combinatorial Logic: Mux

- 4-to-1 Mux out of 2-to-1 muxes
- (one possible solution)



General Questions & Answers

- Take questions/requests from students