**Q1)** Comparing the 2 C programs below, how are they different?

**Q2a)** You are provided with the RTL for the program on the left. Write the assembly code for this program.

**Q2b)** Which instructions leverage static computation (computed at compile time) and which require dynamic computation (computed at runtime)

**Q3a)** Write the RTL code and Assembly for the second C program.
Omit the line: `b = malloc (10*sizeof(int));`

**Q3b)** Which instructions leverage static computation (computed at compile time) and which require dynamic computation (computed at runtime)

**Q4)** What are the differences between the 2 versions?

**Q5)** What are the pros/cons of statically and dynamically allocated arrays

**A2 & A4)**

```
int a;
int b[10];

void foo () {
  b[a] = a;
}
```

RTL

```
r[0]              ← 0x1000

r[1]              ← m[r[0]]

r[2]              ← 0x1004

m[r[2]+r[1]*4]    ← r[1]
```

```
int  a;
int* b;

void foo () {
  b = malloc (10*sizeof(int));
  b[a] = a;
}
```

RTL

**Q6)** Given the following code, what do you think the output will look like? Specifically, which of the printed values will be equal? NOTE: %p is the format specifier for a pointer.

```
int a_static[10];
int* a_dynamic;

void foo () {
    a_dynamic = malloc(10 * sizeof(int));

    printf("address of a_static:  %p\n",   &a_static);
    printf("address of a_static[0]:%p\n",  &a_static[0]);

    printf("address of a_dynamic:  %p\n",  &a_dynamic);
    printf("address of a_dynamic[0]:%p\n", &a_dynamic[0]);
}
```

**Q7)** What will happen when you compile/run the following two programs? Java (left), C (right)

```
public class Foo {
  static int a;
  static int b[] = new int[10];

  void foo () {
   b[20] = 100;
  }
}
```

```
int  a;
int* b;

void foo () {
  b = malloc (10*sizeof(int));
  b[20] = 100;
}
```

Answer the following questions as if you would be adding the code to the end of the following program:

```
int i;
int* a_dynamic;

void foo () {

    a_dynamic = malloc(10 * sizeof(int));

    // fill a_dynamic with values: [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
    for(i=0; i<10; i++)
        a_dynamic[i] = i*10;

}
```

**Q8)** What will the output of these two statements look like with respect to each other?

```
    printf("address of a_dynamic:         %p\n", &a_dynamic);
    printf("address of a_dynamic[0]:      %p\n", &a_dynamic[0]);
    printf("address of a_dynamic[1]:      %p\n\n", &a_dynamic[1]);
```

**Q9)** Given the output of the above statements, what will the output of the following statement be?

```
printf("address of a_dynamic[1] + 1:    %p\n\n", &a_dynamic[1] + 1);
```

**Q10)** Given the output of the above statements, what will the output of the following statement be.
HINT - I am subtracting two pointers. From above we know: Pointer arithmetic takes into account the size of the type of data that pointer is pointing to.

```
printf("&a_dynamic[7]-&a_dynamic[2]:  %lu\n\n", &a_dynamic[7] - &a_dynamic[2]);
```

**Q11)** What will the output of these two statements look like with respect to each other?

```
printf("value of a_dynamic:           %p\n", a_dynamic);
printf("address of a_dynamic[0]:      %p\n\n", &a_dynamic[0]);
```

**Q12)** Given what we now know about pointer arithmetic and the output of the above statements, what will the output of the following statement be

```
printf("value of a_dynamic + 1:       %p\n\n", a_dynamic + 1);
```

**Q13)** What will the output of the following statement be

```
printf("(a_dynamic+7)-(a_dynamic+2):  %lu\n\n", (a_dynamic+7) - (a_dynamic+2));
```

**Q14)** RECALL: a_dynamic contains values: `[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]`
NOTE: when we apply the & to a pointer variable (p) (ie. &p), we read it as (address of p)
NOTE: when we apply the * to a pointer variable (p) (ie. *p), we read it as (value that p points to)
What will the output be after these statements?

```
printf("a_dynamic[0], value a a_dynamic[0]: %d\n", a_dynamic[0]);
printf("*a_dynamic,value that a_dynamic points to: %d\n\n", *a_dynamic);
```

**Q15)** What will the output be after these statements?

```
*a_dynamic = 10;
printf("*a_dynamic, value that a_dynamic points to: %d\n\n", *a_dynamic);
```

**Q16)** What will the output be after these statements?

```
*(a_dynamic + 3) = -1;
printf("a_dynamic[0], value at a_dynamic[0]:              %d\n", a_dynamic[0]);
printf("a_dynamic[3], value at a_dynamic[3]:              %d\n", a_dynamic[3]);
printf("*(a_dynamic+3), value that (a_dynamic+3) points to: %d\n\n", *(a_dynamic+3));
```

**Q17)** What will the output be after these statements?

```
a_dynamic++;  // equivalent code a_dynamic = a_dynamic+1
printf("a_dynamic[0], value at a_dynamic[0]:              %d\n", a_dynamic[0]);
printf("*a_dynamic, value that (a_dynamic) points to:   %d\n\n", *a_dynamic);
```

**Q18)** What will the output be after these statements?

```
a_dynamic += 2;  // equivalent code a_dynamic = a_dynamic+2
printf("a_dynamic[0], value at a_dynamic[0]:              %d;\n", a_dynamic[0]);
printf("*a_dynamic, value that (a_dynamic) points to:   %d\n\n", *a_dynamic);
```

**Q19)** how many elements in a_dynamic now?

**Q20)** What are the elements in a_dynamic now?

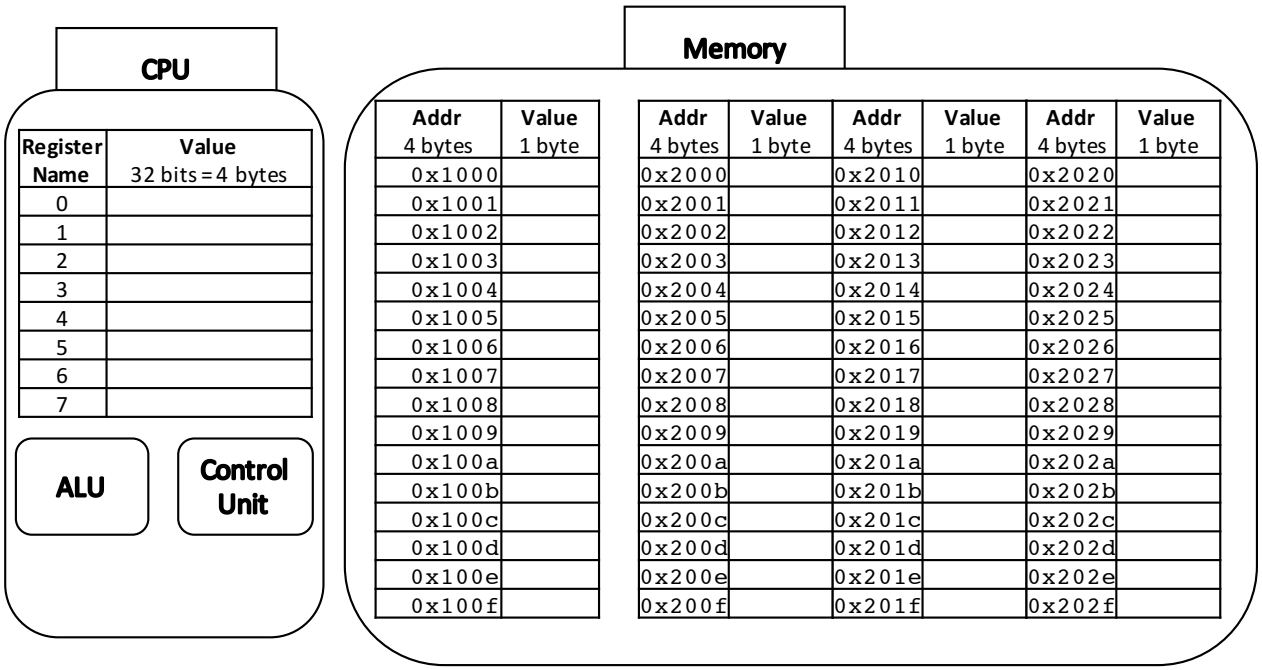
**Q21)** If I printed the value at index 6, what would print?

**Q22)** What would the printf look like if I was printing the value at index 6 using `[ ]` notation?

**Q23)** How would I print the value now at index 6 using pointer arithmetic?

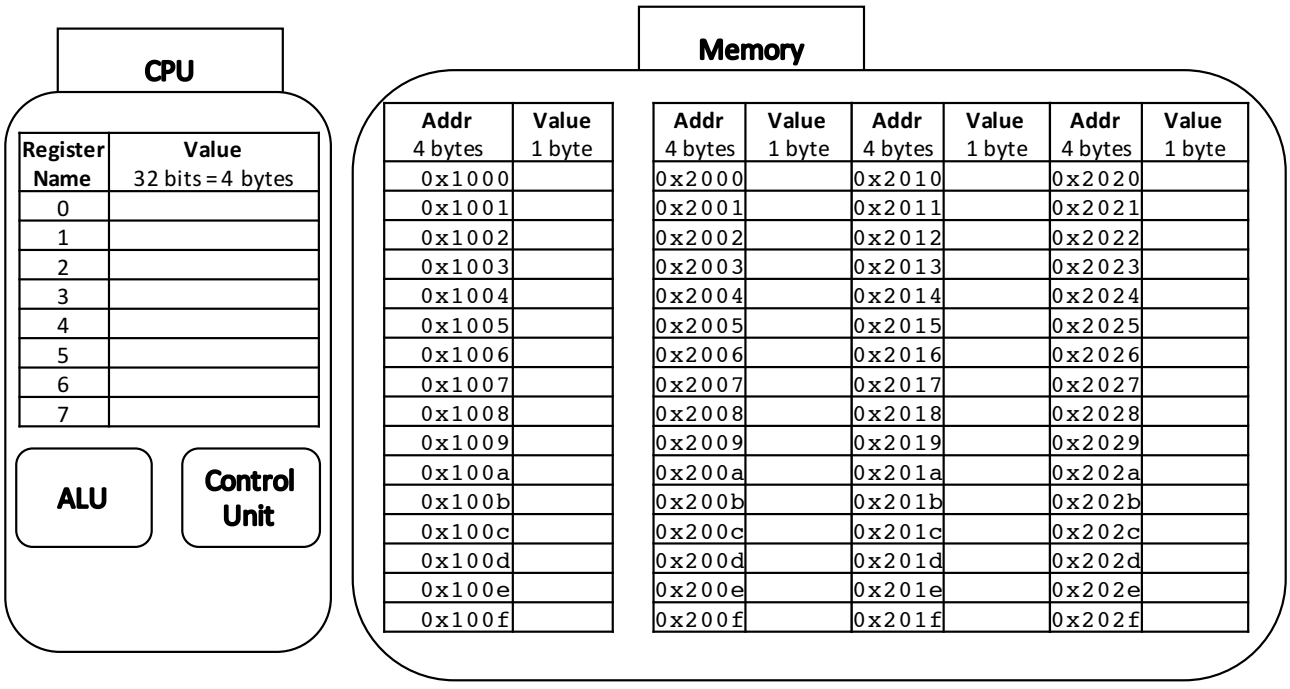
**Q24)** What will the output of this statement be?

```
printf("*a_dynamic+6, value at a_dynamic+6: %d\n\n", *a_dynamic+6);
```




```
int  a;
int b[10];

void foo () {
  b[a] = a;
}
```




```
int  a;
int* b;

void foo () {
  b = malloc (10*sizeof(int));
  b[a] = a;
}
```

## CPU

| Register Name | Value 32 bits = 4 bytes |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

**ALU**

**Control Unit**

## Memory

| Addr 4 bytes | Value 1 byte |
|---|---|
| 0x1000 | |
| 0x1001 | |
| 0x1002 | |
| 0x1003 | |
| 0x1004 | |
| 0x1005 | |
| 0x1006 | |
| 0x1007 | |
| 0x1008 | |
| 0x1009 | |
| 0x100a | |
| 0x100b | |
| 0x100c | |
| 0x100d | |
| 0x100e | |
| 0x100f | |

| Addr 4 bytes | Value 1 byte | Addr 4 bytes | Value 1 byte | Addr 4 bytes | Value 1 byte |
|---|---|---|---|---|---|
| 0x2000 | | 0x2010 | | 0x2020 | |
| 0x2001 | | 0x2011 | | 0x2021 | |
| 0x2002 | | 0x2012 | | 0x2022 | |
| 0x2003 | | 0x2013 | | 0x2023 | |
| 0x2004 | | 0x2014 | | 0x2024 | |
| 0x2005 | | 0x2015 | | 0x2025 | |
| 0x2006 | | 0x2016 | | 0x2026 | |
| 0x2007 | | 0x2017 | | 0x2027 | |
| 0x2008 | | 0x2018 | | 0x2028 | |
| 0x2009 | | 0x2019 | | 0x2029 | |
| 0x200a | | 0x201a | | 0x202a | |
| 0x200b | | 0x201b | | 0x202b | |
| 0x200c | | 0x201c | | 0x202c | |
| 0x200d | | 0x201d | | 0x202d | |
| 0x200e | | 0x201e | | 0x202e | |
| 0x200f | | 0x201f | | 0x202f | |


## CPU

| Register Name | Value 32 bits = 4 bytes |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

**ALU**

**Control Unit**

## Memory

| Addr 4 bytes | Value 1 byte |
|---|---|
| 0x1000 | |
| 0x1001 | |
| 0x1002 | |
| 0x1003 | |
| 0x1004 | |
| 0x1005 | |
| 0x1006 | |
| 0x1007 | |
| 0x1008 | |
| 0x1009 | |
| 0x100a | |
| 0x100b | |
| 0x100c | |
| 0x100d | |
| 0x100e | |
| 0x100f | |

| Addr 4 bytes | Value 1 byte | Addr 4 bytes | Value 1 byte | Addr 4 bytes | Value 1 byte |
|---|---|---|---|---|---|
| 0x2000 | | 0x2010 | | 0x2020 | |
| 0x2001 | | 0x2011 | | 0x2021 | |
| 0x2002 | | 0x2012 | | 0x2022 | |
| 0x2003 | | 0x2013 | | 0x2023 | |
| 0x2004 | | 0x2014 | | 0x2024 | |
| 0x2005 | | 0x2015 | | 0x2025 | |
| 0x2006 | | 0x2016 | | 0x2026 | |
| 0x2007 | | 0x2017 | | 0x2027 | |
| 0x2008 | | 0x2018 | | 0x2028 | |
| 0x2009 | | 0x2019 | | 0x2029 | |
| 0x200a | | 0x201a | | 0x202a | |
| 0x200b | | 0x201b | | 0x202b | |
| 0x200c | | 0x201c | | 0x202c | |
| 0x200d | | 0x201d | | 0x202d | |
| 0x200e | | 0x201e | | 0x202e | |
| 0x200f | | 0x201f | | 0x202f | |