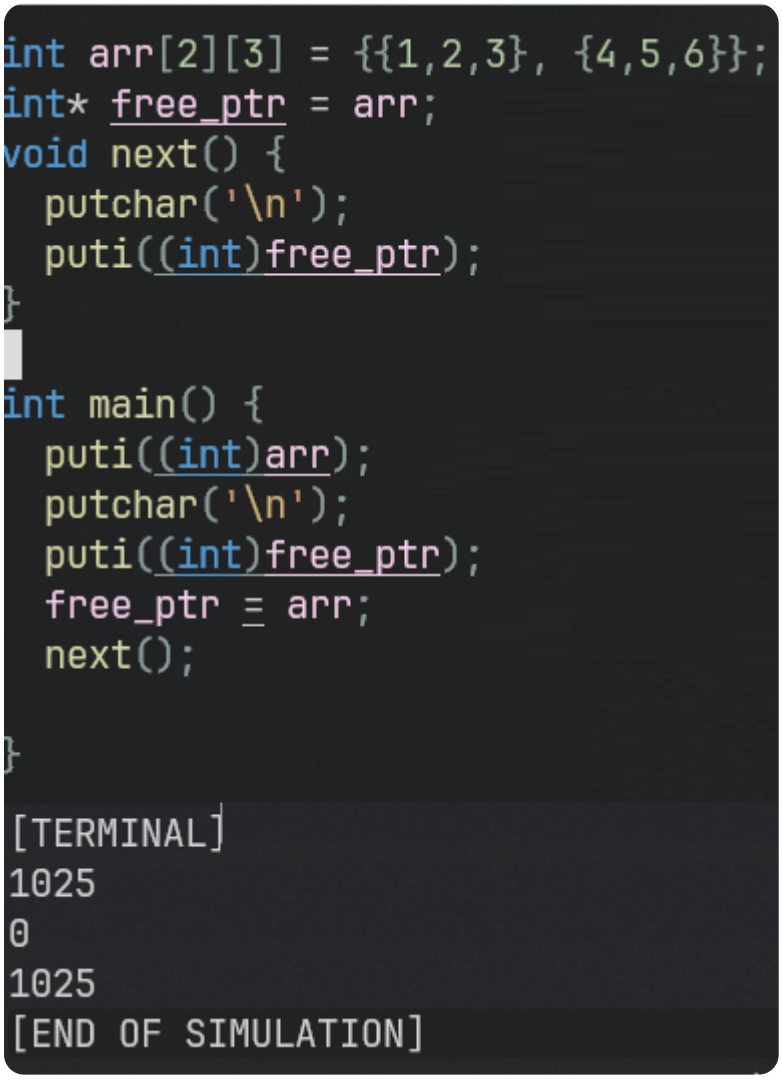
Poxim Backend Details

For a comprehensive understanding of what is possible and correctly implemented, please refer to the `examples/main.c` file in the repository.

Not Implemented

The following list outlines what is confirmed as not implemented:

- **Support for Non-32-bit Basic Data Types:** Basic data types such as `short`, `char`, `float`, `double`, and `long long` are not supported. However, you can use `char` pointers since all pointers are 4 bytes. The backend supports `structs` and `arrays`, but they have certain limitations.
- **Global Assignment to Global Assignment:** Assigning one global variable to another outside of a function is incorrect. This assignment should only occur within a function. Otherwise, it will generate incorrect code. For example:



Note that `free_ptr` only changes when the assignment is made within a function. Furthermore, changing global functions after relocation is a complex testing issue.

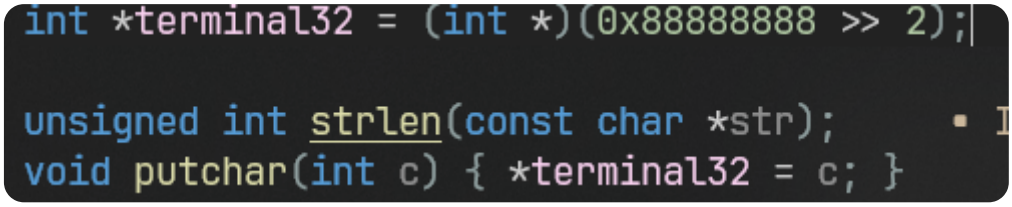
- **Returning a `struct` from a Function:** Returning a `struct` from a function is not supported because it would require implementing the equivalent of `ret n` in x86.
- **Pointer Struct Dereferencing (`*struct_ptr->*`):** Passing a `struct` by pointer has problems when using the `->` operator. This is due to pointer arithmetic in this implementation, as explained in the **Structs** section.

Gotchas

These are common surprising errors that might occur:

Pointers

- **Pointer Shifts:** All pointers are shifted right by two. For instance, the `terminal32` pointer is used for writing to the terminal:



  This shift is due to how load and store operations work in Poxim, requiring 4-byte alignment. Attempting to read a misaligned pointer results in Undefined Behavior.

Pointer Arithmetic

- **Pointer Arithmetic:** Pointers are shifted, so adding one to a pointer is equivalent to adding 4 in conventional C. In this backend, pointer arithmetic only works for pointers that point to data of exactly 4 bytes. For example, if you have a pointer to a vector of 2 `ints` and add 1 to it, you'd expect it to increment the pointer by 8 bytes. However, it increments by 4 and then shifts by two. For example:

```c
struct vec {
  int x,y;
};

int main() {
  struct vec v;
  struct vec* vec_ptr = &v;
  puti((int)vec_ptr);
  putchar('\n');
  puti((int)(vec_ptr + 1));
}
```

This code generates two values that are 1 apart:

```
8185
8186
[END OF SIMULATION]
```

Recommendation: Only use array indexing or pointer arithmetic for data of 4 bytes.

Multidimensional Arrays

- **Multidimensional Array Indexing:** Although you can define multidimensional arrays, any indexing of that array should be done as a pointer to an `int` (`int*`). Using the multiple index syntax results in incorrect machine code due to pointer arithmetic:

```c
int main() {
  int arr[2][3] = {{1,2,3}, {4,5,6}};
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
      puti(arr[i][j]); // Wrong
      putchar('|');
      puti((int)*(arr[i*3 + j])); // Right
      putchar('|');
      puti((int)**(arr + i*3 + j));  // Right
      putchar(' ');
    }
    putchar('\n');
  }

}
```

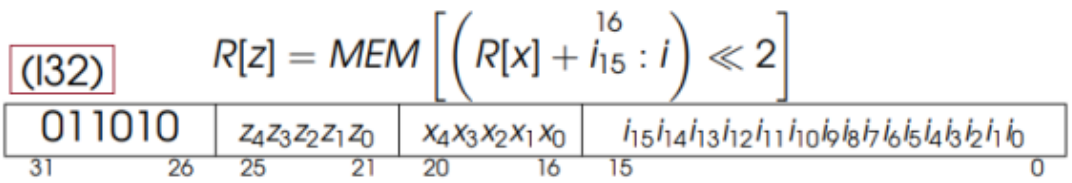
```
[TERMINAL]
1|1|1 2|2|2 3|3|3
2|4|4 3|5|5 4|6|6

[END OF SIMULATION]
```

Recommendation: Always use a pointer to `int` for now.

Immediates as Index

- **Immediates as Index:** Only 16 bits are allowed for immediates, and they are always interpreted as signed integers for simplicity in the machine code generation. For instance:



The code generates a negative index because i_{15} is 1, interpreting it as negative number:

```
int main() {
  int arr[3] = {1,2,3};
  int a = ((arr[(unsigned int)0xfff3]));
```

Resulting in the following code:

```
808:    68 27 ff f1       l32    r1, [r7-15]<<2
```

The same applies to pointer arithmetic. If the immediate exceeds 16 bits, an assert is generated rather than a user-level error.

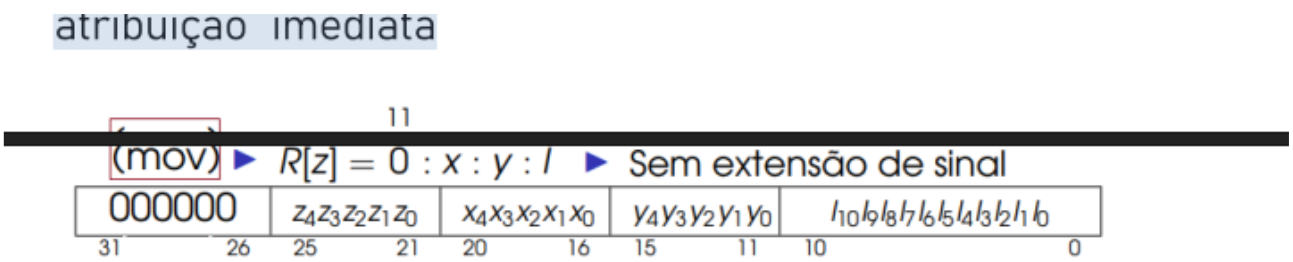
Recommendation: Use signed integers for everything and integer indices of a maximum of 16 bits.

Shifts

- **Shifts:** Only shifts of up to 31 are allowed because it's the maximum needed for a 32-bit integer.

Immediates for Integers

- **Immediates for Integers:** Immediates for integers can only have 20 bits for simplicity in machine code generation. Any immediate exceeding this limit will be truncated:



So code like this will silently truncated to the last 20 bits of that integer

```
int cafe = 0xcafebabe;
```

Resulting in this

```
04 3e ba be       movs   r1, -83266
```

Implemented and Tested

Functions

- **Indirect Function Calls**
- **Storing Function Pointers**
- **Passing Functions Pointers to other functions**

see `examples/main.c` for a use of that, in section array.

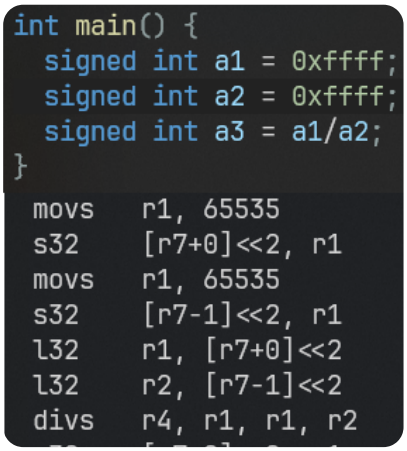
Arithmetic

- Unsigned Division

```
unsigned int a1 = 0xffff;
unsigned int a2 = 0xffff;
unsigned int a3 = a1/a2;
```

```
movs   r1, 65535
s32    [r7-15]<<2, r1
movs   r1, 65535
s32    [r7-16]<<2, r1
l32    r1, [r7-15]<<2
l32    r2, [r7-16]<<2
div    r4, r1, r1, r2
```

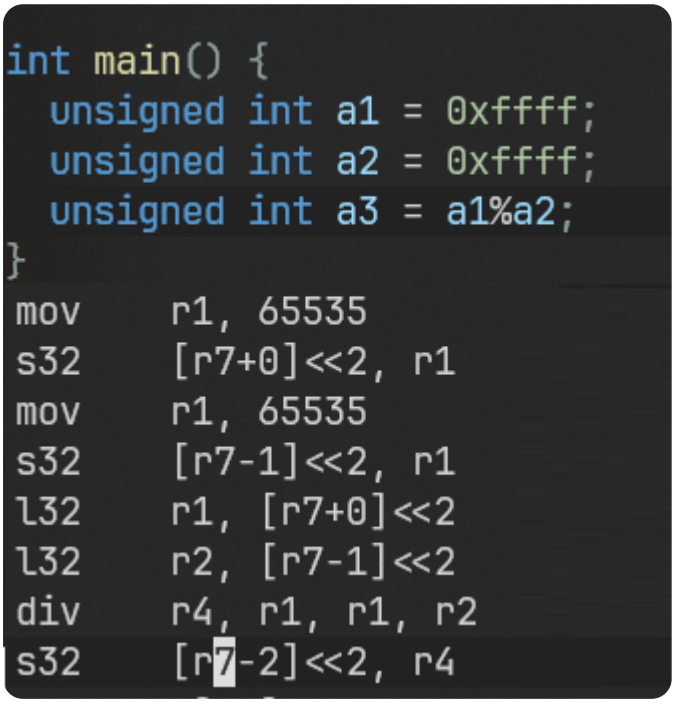
- Signed Division



```
int main() {
  signed int a1 = 0xffff;
  signed int a2 = 0xffff;
  signed int a3 = a1/a2;
}
```

```
movs   r1, 65535
s32    [r7+0]<<2, r1
movs   r1, 65535
s32    [r7-1]<<2, r1
l32    r1, [r7+0]<<2
l32    r2, [r7-1]<<2
divs   r4, r1, r1, r2
```

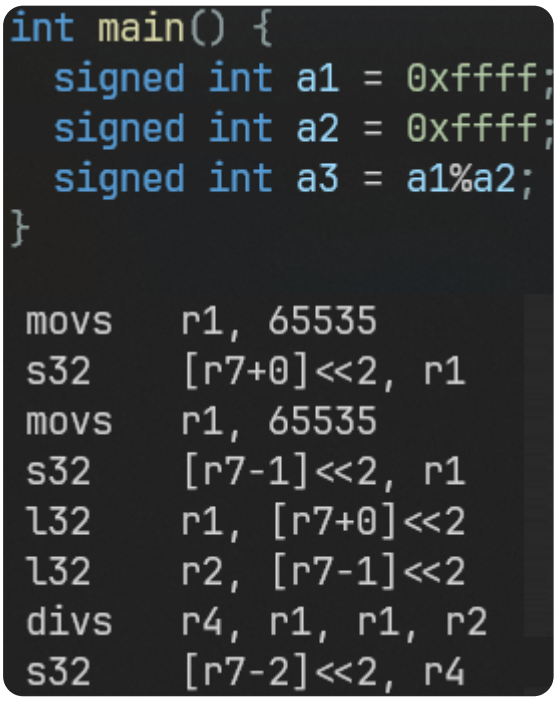
- Unsigned Mod Notice the same instructions as div, but the very next instruction stores r4 (refer to Poxim ISA)



```
int main() {
  unsigned int a1 = 0xffff;
  unsigned int a2 = 0xffff;
  unsigned int a3 = a1%a2;
}
```

```
mov    r1, 65535
s32    [r7+0]<<2, r1
mov    r1, 65535
s32    [r7-1]<<2, r1
l32    r1, [r7+0]<<2
l32    r2, [r7-1]<<2
div    r4, r1, r1, r2
s32    [r7-2]<<2, r4
```

- Signed Mod Is the same as before but divs and movs instructions instead



These operations are supported for integer

- Multiplication `*`
- Division `/`
- Remainder `%`
- Additiona `+`
- Subtraction `-`
- Bitwise AND, OR, NOT, XOR `& | ~ ^`
- Left and Right Shifts (`>>` and `<<`)

However, XOR is not thoroughly tested but should work fine.

- **Unary Increment and Decrement**: `x++` and `x--`
- **Local Pointer Arithmetic**: Only for pointers of 4 bytes of data is supported. The "pointer of" operator, like `int ptr = &a`, is supported. Note that all pointers must be aligned to 4 bytes due to the `s32` instruction.
- **Global Variable Initialization | Assignment**: You can intitialize a global variable to a constant value. However, initializing the value of globals by using another global is undefined behavior. It might work or it might be incorrect. In general, it's better to use pointers, integers, or

structs locally in a function. If you need to change a global value, do it inside of a function.

- **Supported Pointer Types**: Function pointers, `void*` pointers, and `int*` pointers were tested. However, any pointer should behave as a 32-bit value. With the usual gotcha that it's always interpreted as a pointer to integer, sorry about that :).

Logical Operations

- **All Logical Operators:** `!`, `&&`, `||`, `>=`, `<=`, `>`, `<`,

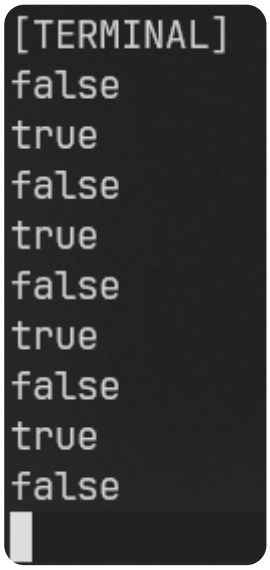
```c
#define MAX_TEST 50
int main(void) {
  int bools[MAX_TEST] = { 0 };

  int a = 1;
  int b = 2;
  int c = 3;
  int d = 4;
  int e = 5;
  int f = 6;

  int idx = 0;
  bools[idx++] = (a > b);                    // false
  bools[idx++] = (a < b) && b + 2 == 4;      // true
  bools[idx++] = (a < b) && b + 2 != 4;      // false
  bools[idx++] = (b >= b) || e != 5 ;        // true
  bools[idx++] = (b >= b) && e != 5 ;        // false
  bools[idx++] = (e != 5 || d != 4 || 1);    // true
  bools[idx++] = (e != 5 || d != 4 && 1);    // false
  bools[idx++] = (e < 5 || d < 4 || a < b);  // true
  bools[idx++] = !(e < 5 || d < 4 || a < b); // false

  for (int i = 0; i < idx && i < MAX_TEST; i++) {
    putbool(bools[i]);
    putchar('\n');
  }

  return 0;
}
```



- **Storing Boolean Results**: This is equivalent to `setxx` in x86, but the backend does not have an equivalent instruction so it handles the logic to conditionally move or set when storing using other primitive instructions.

Structs

- **Definition and Initialization of Structs**
- **Passing Structs to Functions**: Both by pointer and by value are allowed.

By Pointer:

Passing by pointer has the same problem as described in the Pointer Arithmetic section. Accessing the elements of a struct by pointer will be incorrect unless all members of the struct are 4 bytes. To access elements correctly, you'd have to know the offset of each member. Here's

a Hacky way to do it for the following struct C:

```c
typedef struct {
  int a1, a2;
} A;

typedef struct {
  A a;
  int b1, b2, b3;
} B;

typedef struct {
  B b;
  int c1, c2, c3, c4;
} C;
```

```c
// Works, but hack
void init_c(C* c){
    *(int*)c = 1;
    *(int*)(c+1) = 2;
    *(int*)(c+2) = 3;
    *(int*)(c+3) = 4;
    *(int*)(c+4) = 5;
    *(int*)(c+5) = 6;
    *(int*)(c+6) = 7;
    *(int*)(c+7) = 8;
    *(int*)(c+8) = 9;
}
```

```c
int main() {
  puts("Main Struct C:\n");
  C c;
  init_c(&c);
  print_c(c);
}
```

```
[TERMINAL]
Main Struct C:
123456789
[END OF SIMULATION]
```

The Wrong way would be doing something like this:

```
 // Does not Work correctly Don't use it
void wrong_init_c(C* c){
    c->b.a.a1 = 1;
    c->b.a.a2 = 2;
    c->b.b1 = 3;
    c->b.b2 = 4;
    c->b.b3 = 5;
    c->c1 = 6;
    c->c2 = 7;
    c->c3 = 8;
    c->c4 = 9;
}
```

By Value:

Passing by value should always result in correct code. It does not rely on pointer arithmetic but rather on the stack argument passing system. This works:

```
void print_c(C c){
    puti(c.b.a.a1);
    puti(c.b.a.a2);
    puti(c.b.b1);
    puti(c.b.b2);
    puti(c.b.b3);
    puti(c.c1);
    puti(c.c2);
    puti(c.c3);
    puti(c.c4);
};
```

This also works :

```c
int main() {
  int cafe = 0xcafebabe;
  puti(cafe);
  puts("Main Struct C:\n");
  C c;
  c.b.a.a1 = ~1;
  c.b.a.a2 = 2;
  c.b.b1 = 3;
  c.b.b2 = 4;
  c.b.b3 = 5;
  c.c1 = 6;
  c.c2 = 7;
  c.c3 = 8;
  c.c4 = 9;
  print_c(c);
  int a = 2, b = 3;
  puti(a ^ b);
}
```

```
[TERMINAL]
Main Struct C:
123456789
[END OF SIMULATION]
```

That's it for now, but this document might be revised in the future if i remember any thing else that's important. The final recommendation is to look at the `examples` folder specially `examples/main.c` where we have `#ifdefs` that might elucidate the difference between **GNU** compilation and **TCC Poxim** compilation. This ifdef were made exactly where they differ, especially in pointer arithmetic 💔 Ty for the wonderful classroom and semester cheers.