---

1. Language Overview

- **Statically typed**, **stack-only** programming language (PL).
- Compiles to a custom **register-based assembly** (12-bit and 6-bit variants).
- **Single stack frame** per thread; no heap, no recursion, no VLAs, no `alloca`.
- All values live on the stack; pointers/references to stack slots are allowed but cannot escape their frame.
- **Functions** take one argument and return one value (both can be compound types: tuples or vectors).
- **Key syntax patterns**: comma-separated `key:value` lists enclosed in block markers.
- **RHS expressions in `SET ... WITH` blocks are evaluated in parallel**, unless they contain impure operations (which is a compile-time error).
- **Operators** can be applied to tuples or vectors if compatible:
- Vectors: same length and compatible base types
- Tuples: same fields with compatible types

1.1. Core PL Syntax

```
VARIABLES
  name1: type1
, name2: type2
, ...
END
```

```
TYPEDEF
  alias1: existing_type1
, alias2: existing_type2
END
```

```
SET
  lval1: rval1
, lval2: rval2
WITH
  symbol1: rval3
, symbol2: rval4
END
```

```
FUNCTION_HEADERS
  name1: in_type1:out_type1
```

```
, name2: in_type2:out_type2
END
```

```
FUNCTION_BODIES
  name1: BEGIN
            ...statements...
         END
, name2: instruction2
END
```

```
EXPORT
  out_name: in_name
END
```

```
IMPORT
  FROM module_name
    in_name: out_name
END
```

```
SUM_TYPE
  OptionA: TypeA
, OptionB: TypeB
END
```

```
MATCH value WITH
  OptionA:x: stmtA
, OptionB:y: stmtB
END
```

---

2. Assembly Language (12-bit variant)

2.1. Instruction Format

- **12 bits total**:
- **4 MSBs**: Command family
- **8 LSBs**: Payload (arguments, type codes, etc.)

2.2. Registers

- **Three core registers**:

- `left` — primary pointer/address
- `right` — secondary pointer/address
- `param1` — auxiliary (e.g., passing syscall parameters)
- **Optional fourth register**: `result`
- **Zeroing behavior**: On jumps, calls, syscalls, and when selecting a register.

2.3. Register Input Commands

- **Push immediate**: pushes a byte into the currently selected register.
- 12-bit version: 8-bit immediates
- 6-bit version: 4-bit immediates

2.4. ALU Operations and Casting

- ALU is **stateful**: holds a current data type (u8, i8, u16, i16, u32, i32, f32).

- **Cast/Copy Command** takes two 4-bit type codes (`source`, `destination`):

- **Case 1**: `source == destination` and `left == right` → default operation only.

- **Case 2**: `source == destination`, `left != right` → copy `sizeof(type)` bytes from `right` to `left`, then default.

- **Case 3**: `source != destination` → cast from `source` to `destination`, store at `left`, then default.

- **Default operation** (performed after any copy/cast):

```
*(destination*)left = *(source*)right;
left  += sizeof(destination);
right += sizeof(source);
type_of_ALU = destination;
```

- **Vectorized operators** also use `left` / `right` cursor advancement, enabling efficient tuple/vector processing.

2.5. Memory Model

- **Stack layout per function**:
- Address **0**: return value
- Next addresses: function parameter, then local variables
- **Address encoding**:
- LSB = 0 → stack-local address (offset from frame start)
- LSB = 1 → module-level static/global variable

2.6. Indirection and Pointers

- **Indirect load/store**: operate on the memory address stored *at* the address in a register.
- No heap; all indirection is within the stack or static data.

2.7. Control Flow

- **Relative jumps**: offset from function's first instruction.
- Jump to **0** → restart function.
- Jumps reset registers (zero them).

2.8. Function Calls

- **Syscall-based call**:
- Place **magic number** in a designated register.
- Provide return-value address in another register.
- Other regs (param1) for argument info if needed.

---

3. Executable Format and Linking

3.1. Global Tables

1. `global_to_local`: maps each **global_id** (implicit index) → `(module_id, local_id)`
2. `module_metadata`: rows keyed by **module_id** (implicit) with columns:
3. `first_extern_function`: index of first external function
4. `module_location`: file offset
5. `tables_length`, etc.

3.2. Per-Module Tables

1. `function_ends`: end offsets of each function (implicit function_id ordering). → can derive start offset from previous end.
2. `extern_to_global`: for each external index (implicit), maps → `global_id`.

3.3. Function Magic Number Resolution

```
if (magic_number < first_extern_function) {
  module_id   = current_module_id;
  function_id = magic_number;
} else {
  extern_id    = magic_number - first_extern_function;
  global_id    = extern_to_global[extern_id];
  module_id    = global_to_local[global_id].module;
  function_id  = global_to_local[global_id].function;
```

```
}
// Use (module_id, function_id) to look up jump offset
```

- **System module** ID is `0xFFFF` for syscalls and runtime services.
- Enables **compact** function references (small magic numbers) with **fast native lookups**.

Next steps: - Define **function-module magic number encoding** details. - Describe **executable file layout**: header, table sections, code segments, static data. - Outline **compiler** phases: parsing, type-checking, IR lowering, assembly emission, linking.