DEVELOPMENT OF A BITROUT COMPILER, A PROGRAMMING LANGUAGE FOR A PRIMITIVE ARCHITECTURE

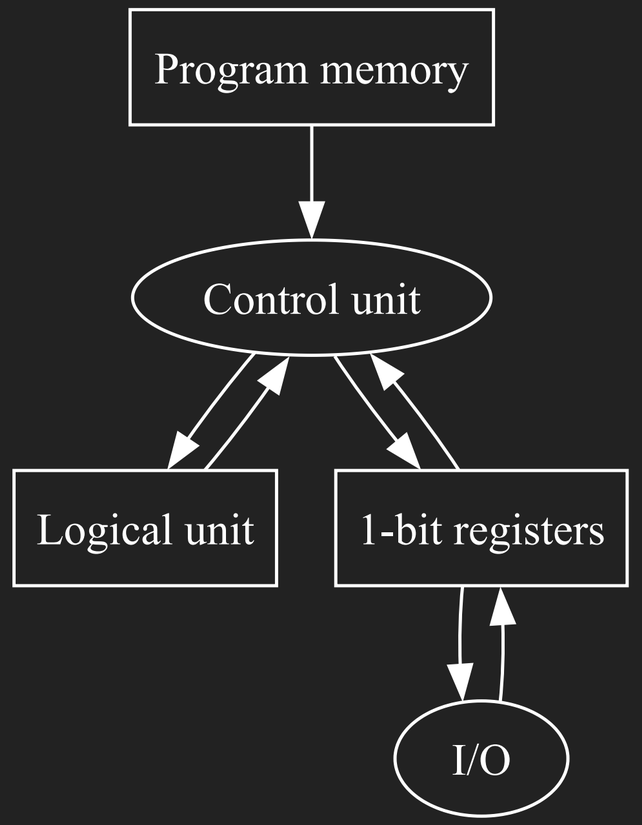
ADRIAN SEBASTIAN ŠIŠKA MENTOR: ALEŠ VOLČINI

MIDEM 2023 Conference, September 27th – 29th, 2023, Lipica

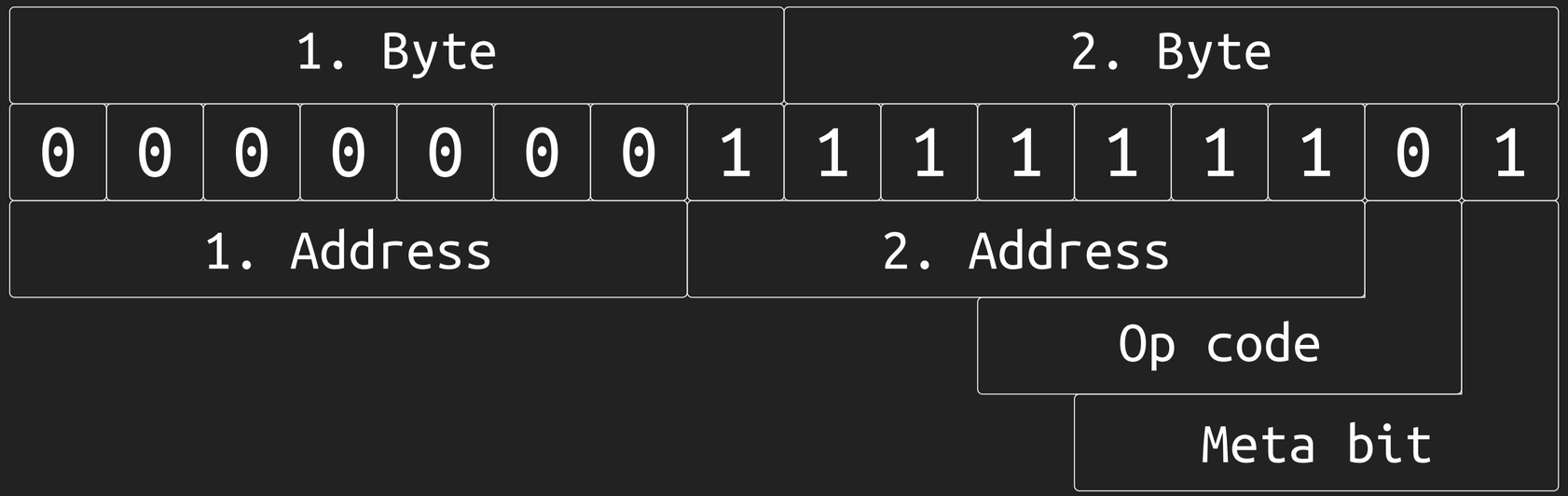
1. QUICK RECAP

- MUHI architecture

1.1. MUHI



Instruction:



Registers:

1.1.1. INSTRUCTIONS

- 1-bit
  - Nand, Xor
- 16-bit
  - copy, load

2. MOTIVATION

- Code optimisation

2.1. HASSLE OF A BAD ASSEMBLER

```
.org 4
%include "std.asm"

%macro print2 2
    c16 %1, %2, 1
    inc16 %2
    //exit
    set_out_b %2 + 0 , 0x14
    set_out_b %2 + 1 , 0x14
    set_out_b %2 + 2 , 0x14
    set_out_b %2 + 3 , 0x14
    set_out_b %2 + 4 , 0x14
    set_out_b %2 + 5 , 0x14
    set_out_b %2 + 6 , 0x14
    set_out_b %2 + 7 , 0x14
    set_out_b %2 + 8 , 0x14
    set_out_b %2 + 9 , 0x14
    set_out_b %2 + 10, 0x14
    set_out_b %2 + 11, 0x14
    set_out_b %2 + 12, 0x14
    set_out_b %2 + 13, 0x14
    set_out_b %2 + 14, 0x14
    set_out_b %2 + 15, 0x14
%endm

word:
.db b"Zivjo\n"

main:
    print2 labels["word"]  , 0x15
    print2 labels["word"]+1, 0x15
    print2 labels["word"]+2, 0x15
    exit

init "main"
```

3. THE BITROUT™ LANGUAGE

- keywords: `fn`, `var`, `include`, `if`, `else`
- builtins: `set`, `get`, `x`, `a`, `goto`

3.1. EXAMPLE FUNCTION

```
fn to_hex(in:4b| out:8b) {
    var check;
    leeq in 9 check;
    is in out[-4..];
    if check {
        add "0" out;
    } else {
        add "a" out;
    }
}
```

3.2. FUNCTION ARGUMENTS

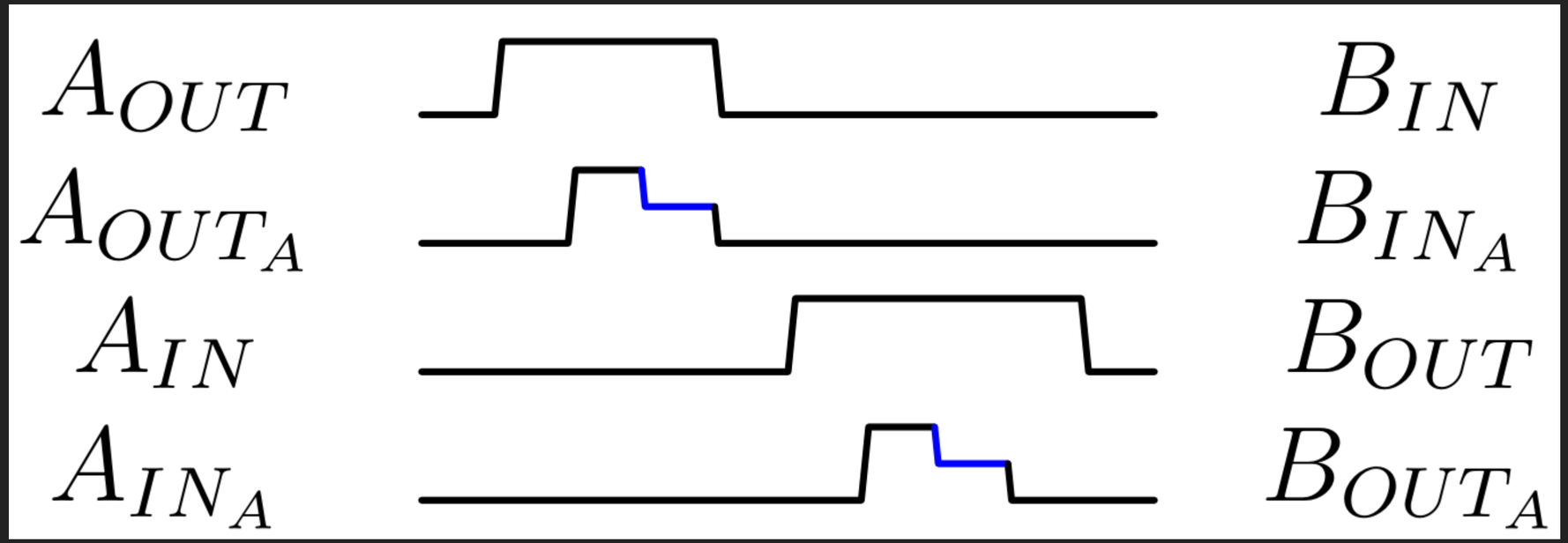
```
fn example(a, b:1, c:2b, d:3r | e, f:1, g:4b, h:1r)
                  ^    ^     ^  ^
                  |    |     |  \_ explicit length
                  |    |     \____ bitwise length
                  |    \__________ bitwise length reversed
                  \_______________ mutability seperator
```

```
fn print(string:8b){...}

{
    print "abc";
}
```

```
fn print(string:8b){...}

{
    print "c";
    print "b";
    print "a";
}
```

3.3. SYSCALLS

- 2 system calls for I/O



3.3.1. GO-LIKE EXCEPTIONS

```
⋮
var E;
syscall1 arg1 E;
set arg2 E;
syscall2 arg3 E;
if E {
    var E2;
    set "Catching exception!" E2;
}
```

3.4. LIMITATIONS

- No runtime recursion ⇒ differnt runtime and comptime

3.4.1. COMPTIME EXAMPLE

```
fn ComptimeIncrement(|A){
    var length;
    lenOf A length;

    var T;
    neq length 0 T;
    if T {
        if A[0]{
          is 0 A[0];
          ComptimeIncrement A[1..];
        } else {
            is 1 A[0];
        }
    }
}
```

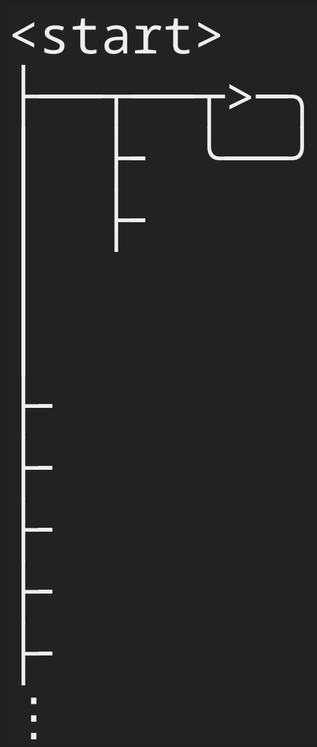
3.4.2. RUNTIME EQUIVALENT CODE

```
fn _inc(|A:1b, Carry:1){
    if Carry {
        is A Carry;
        not A;
    }
}

fn inc(|A) {
    var Carry:1 = 1;
    _inc A Carry;
}
```

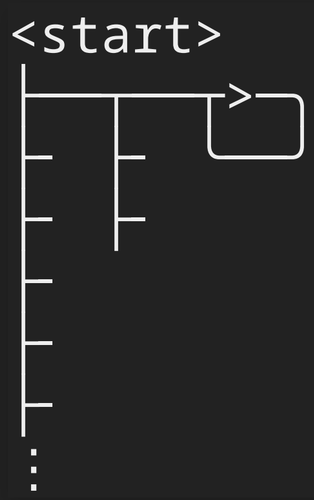
4. COMPILER

1. Lexical analysis
2. Syntactic analysis
3. Length deduction
4. Translation into “branches”
5. Branch reduction (optional)
6. Memory acquisition

4.1. BRANCH REDUCTION

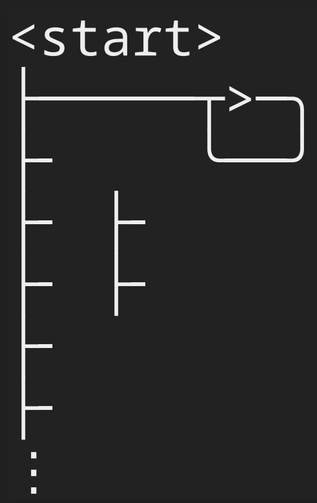


4.1.1. REMOVING REDUNDANT INSTRUCTIONS



-(-x) = x

4.1.2. ELIMINATING DEAD BRANCHES




```
if (false) { => b;
    a;
} else {
    b;
}
```

4.1.3. THE PARTS LEFT BEHIND

- structure
- branch reuse