SUS Language Reference Guide

v0.3.1

Variables & Assignments

simple declaration `int addr`
declaration and assign `float[2] xs = [7.5, -5.0e3]`
later assigns overwrite `xs[1] = 0.0`
immediately visible `float[2] ys = xs  // == [7.5, 0.0]`
unassigned == don't care `int not_written  // == 'x`

state reg declaration `state int cur_idx`
set initial value `initial cur_idx = 0`
update every cycle `cur_idx = (cur_idx + 1) % 10`

declare state register `state int st`
state updates next cycle `st = 12`
not visible yet `int z = st  // == previous st`

compile-time variable `gen int[2] VALS = [3, -7]`
use gen vars anywhere `gen bool[VALS[0]] my_arr`

Sized Integers

wire ints have bounds `int#(FROM: 0, TO: 10) digit = 9`
TO is exclusive `int#(FROM: 5, TO: 6) five = 5`
bounds can be inferred `int sum = digit + five`
`gen bool[10] MY_BOOLS = [...]`
index is boundschecked `bool b = MY_BOOLS[sum]`
gen ints are unbounded `gen int BOOP = -500000000000000`

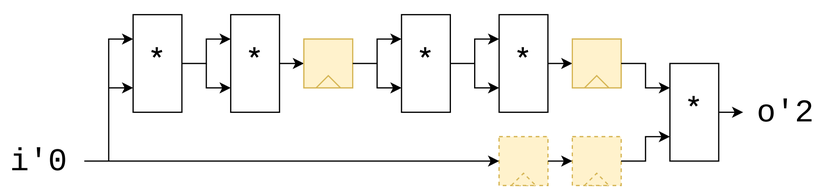
Parameters

module declaration `module m #(TypeParam, int VALUE) { ... }`
constant declaration `const int sizeof#(T) { ... }`
type declaration `struct int #(int FROM, int TO) { ... }`
module param usage `m #(TypeParam: type bool, VALUE: 6) inst`
type param usage `gen int TO = sizeof #(T: type bool[4])`
`gen int FROM = -TO`
constant param usage `int #(FROM: -TO, TO: TO) balanced_octal`
or more compactly `int #(FROM, TO) balanced_octal`

Latency Counting

add pipeline stage
add pipeline stage
'i' gets compensating regs

```
module pow17 : int i -> int o {
    int i2  = i  * i
reg int i4  = i2 * i2
    int i8  = i4 * i4
reg int i16 = i8 * i8
    o   = i  * i16
}
```

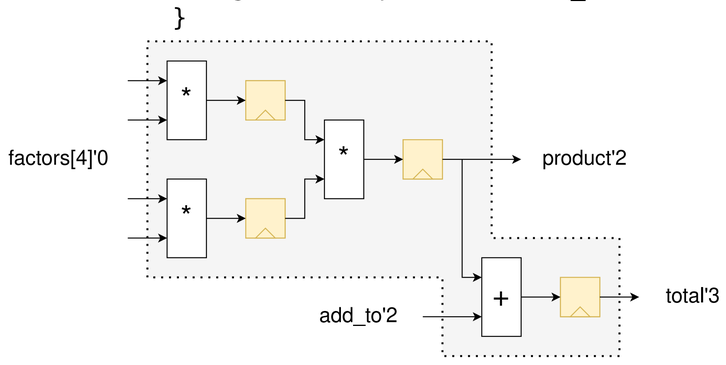


add two latency registers
shorthand decl + regs

```
bool x
reg reg x = true
reg reg reg int v = 6
```

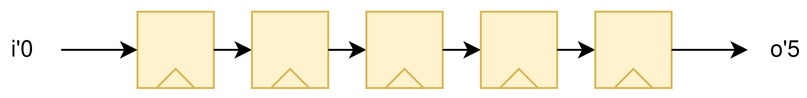
factors'0 inferred
add_to'2 inferred
product'2 inferred
total'3 inferred

```
module example_md :
  int#(FROM: 0, TO: 100)[4] factors,
  int#(FROM: 0, TO: 100) add_to ->
  int product,
  int total {

  reg int mul0 = factors[0] * factors[1]
  reg int mul1 = factors[2] * factors[3]

  reg product = mul0 * mul1
  reg total = product + add_to
}
```



i specified to '0
o specified to '5
forces 5 registers

```
module specified_ports :
  int i'0 ->
  int o'5 {
  o = i
}
```



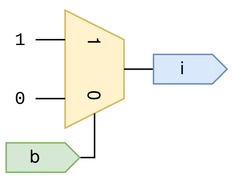
Control Flow

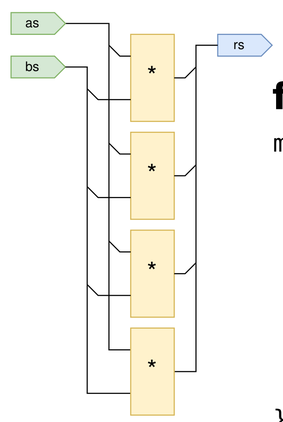
if: compiletime

```
module axi_send #(T) {
  input T data
  axis axis

  if sizeof#(T) <= 32 {
    axis.send_single(data)
  } else if sizeof#(T) <= 64 {
    axis.send_wide(data)
  } else {
    axis.send_burst(data)
  }
}
```

when: runtime

```
module to_int {
  input bool b
  output int i

  when b {
    i = 1
  } else {
    i = 0
  }
}
```





for

```
module vector_mul {
  interface vec_mul:
    float[4] as, float[4] bs ->
    float[4] rs

  gen int LEN = 4
  for int I in 0..LEN {
    rs[I] = fp_mul(as[I], bs[I])
  }
}
```

Module Ports

regular input
regular output

```
module m {
  input float[4] xs
  output float[4] ys
}
```

instantiate module `m m_inst`
assign to input `m.xs = [0.2, 0.3, 0.4, 0.5]`
read from output `float y = m.ys[2]`

"main" interface

```
module fp_add {
  interface fp_add:
    float a, float b -> float c
}
```

inline usage `float x = fp_add(0.2, 0.7)`

action declaration
trigger declaration

```
module safe_divider {
  action divide : int n, int d {
    trigger nonzero : int div

    when d != 0 {
      is_nonzero(n / d)
    }
  }
}
```

trigger trigger
action use

```
safe_divider safe_div
safe_div.divide(5, 2)
```

trigger use and conditional binding

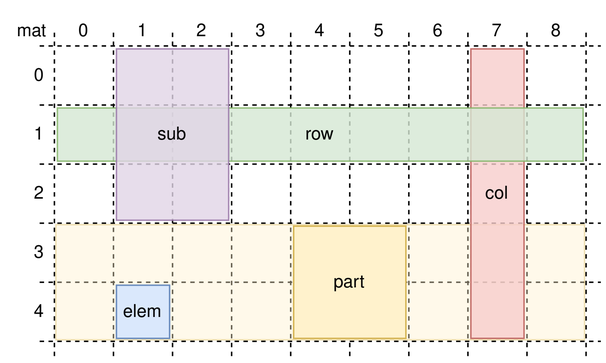
```
when safe_div.nonzero: int div {
  //...
}
```

N-D Arrays & slicing

declaration `input bool[5][9] mat`

indexing `bool elem = mat[1][4]`
access sub-array `bool[5] col = mat[7]`
N-D slicing `bool[9] row = mat[:][1]`
open slice bound `bool[2][2] sub = mat[1:3][:3]`
`input int#(FROM:0, TO: 7) x`
part-select `bool[2] part = mat[x+:2][3:]`

array literal `int[4] values = [1, 2, 3, 4]`
wide operations `int[4] twice = values + values`
`// == [2, 4, 6, 8]`



Builtins

builtin functions

```
gen bool true
gen bool false
gen bool assert #(bool C)
gen int sizeof #(T)
gen int clog2 #(int V)
gen int pow2 #(int E)
gen int pow #(int B, int E)
gen int factorial #(int N)
gen int falling_factorial #(int N, int K)
gen int comb #(int N, int K)
gen int noinfer #(int V)
```

builtin modules

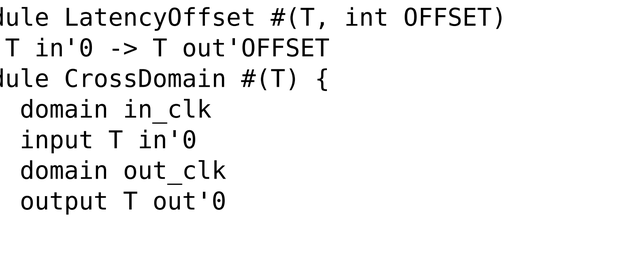
```
module LatencyOffset #(T, int OFFSET)
 : T in'0 -> T out'OFFSET
module CrossDomain #(T) {
    domain in_clk
    input T in'0
    domain out_clk
    output T out'0
}

module transmute_from_bits #(T)
 : bool[sizeof #(T)] bits'0 -> T value'0
module transmute_to_bits #(T)
 : T value'0 -> bool[sizeof #(T)] bits'0
module transmute #(T1, T2): T1 a'0 -> T2 b'0

module unsafe_int_cast
  #(int FROM_I, int TO_I, int FROM, int TO):
    int#(FROM: FROM_I, TO: TO_I) in'0 ->
    int#(FROM, TO) out'0
```

(Latency) Parameter Inference

G unused in ports
N = a.LEN
M ≥ b.TO
M ≥ c.TO + 3

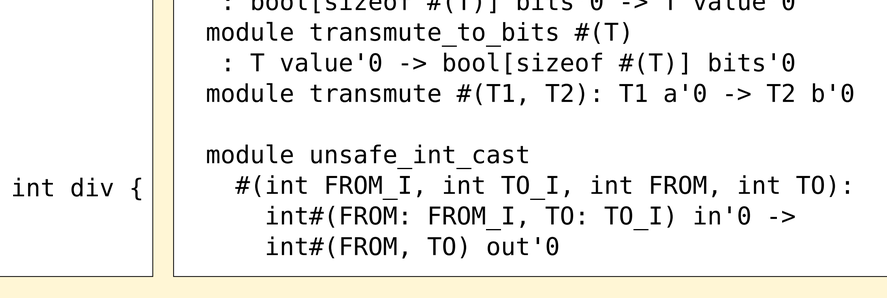
```
module InferMe #(int G, int N, int M, int L) {
  input bool[N] a
  input int#(FROM: 0, TO: M) b
  input int#(FROM: 0, TO: M - 3) c
}
```

G must be given `InferMe#(G: 4) infer_me`
N inferred to 3 `infer_me.a = [true, true, false]`
`int#(FROM: 0, TO: 6) b_in`
`int#(FROM: 3, TO: 5) c_in`
M ≥ 6 `infer_me.b = b_in`
M ≥ 8 `infer_me.c = c_in`
M inferred to 8

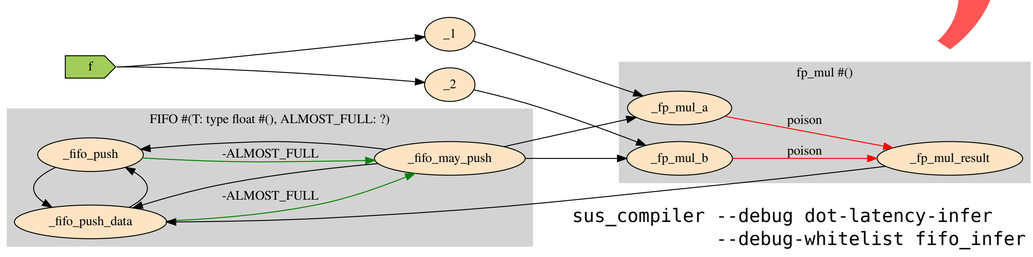



```
module FIFO #(T, int ALMOST_FULL) {
  trigger may_push'-ALMOST_FULL
  action push'0 : T push_data'0 {}
}
```

ALMOST_FULL ≥ 'push - 'may_push
ALMOST_FULL ≥ 'push_data - 'may_push




```
module fifo_infer {
  FIFO fifo
  input float f
  when fifo.may_push {
    fifo.push(fp_mul(f, f))
  }        // fp_mul has 8 regs
}
```




```
sus_compiler --debug dot-latency-infer
             --debug-whitelist fifo_infer
```