HW/SW Co-Design Lab – TIE Documentation

This document provides the basic language constructs of the Tensilica Instruction Extension (TIE). Please consider the *TIE Reference Manual* and the *TIE User's Guide* for detailed information.

Immediate Range

The `immediate_range` construct provides a way to define a range of immediate values that can be used as the operand of an instruction. It is an abstraction that represents the range of legal values that an immediate operand can take. After an immediate range has been declared, it can be used as the type of an argument in an operation statement.

Example:

{-8, -6, -4, 0, 2, 4, 6}
```
immediate_range imrange -8 6 2
```

{0, 1, 2, 3, 4, 5, 6}
```
immediate_range imrange 0 6 1
```

Table

Constant tables are defined with `table` description sections. Constant tables allow the designer to define a hardware table that can be used in any computational expression.

Example:

8-bit wide table with 16 entries
```
table prime 8 16 {
     2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53
}
```

State

Processor states (or registers) are defined with `state` description sections. States behave like variables that hold a value prior to the execution of an instruction. This value can be used for a computation performed by the instruction.

Example:

4 states with different sizes, the last 2 are initialized
```
state DATA 64
state KEYC 32
state KEYD 32 32'h0
state TIME 16 16'hFFFF
```

2 states with different sizes available from C/C++ code
```
state S1 20 add_read_write
state ROUND 50 add_read_write
```

Reading state to variable
```
int s1_state = RUR_S1();
```

Writing variable value to state
```
int s1_state = 5;
WUR_S1(s1_state);
```

Register File

The designer can create additional register files with `regfile` description sections and use these designer-defined register files as operation arguments of TIE instructions.

Example:

Declare a 128-bit wide, 16 entry register file called VR with the short name v
```
regfile VR 128 16 v
```

Define variable in C/C++
```
VR reg128;
```

Furthermore, the predefined address register `AR` or boolean register `BR` can be used.

Operation

The `operation` section describes the name, format, and behavior of a single TIE instruction. It is typically the simplest and most compact way to describe a new instruction.

Example:

The following operation describes an instruction with 2 arguments (AR registers and 1 state). The result is written to the argument specified as `out` and to the state, declared as output, too.

```
state a_state 32
operation ADDACC {inout AR a, in AR b} {out state} {
     assign a = a + b;
     assign a_state = a;
}
```

In C/C++:

```
int acc, a;
ADDACC(acc, a);
```

If the operation has only 1 operand of type `out`, the C/C++ intrinsic has a return value

```
table TBL 32 4 {1, 5, 7, 12}
operation SUBT {out AR x, in TBL y, in TBL z} {} {
     assign x = y - z;
}
```

In C/C++:

```
int diff;
diff = SUBT(5, 12);
```

Schedule

A `schedule` section provides a mechanism for the designer to define the pipeline stages in which input operands of a TIE instruction are read and output operands are computed.

Example:

In the following example, the operation `ADD16` would be executed in 1 clock cycle. However, the schedule `add16_sched` causes the operation to read a and b in cycle 1 and to write the result to `sum` in cycle 2.

```
operation ADD16 {out AR sum, in AR a, in AR b} {} {
     wire [15:0] t0 = a[15: 0] + b[15: 0];
     wire [15:0] t1 = a[31:16] + b[31:16];
     assign sum = {t1, t0};
}

schedule add16_sched {ADD16} {
     use a 1;
```

```
    use b 1;
    def sum 2;
}
```

Instruction Format and Slot Opcodes

Instruction formats can be viewed as templates for specifying which operations can be grouped (packed) into a single FLIX/VLIW instruction. The `slot_opcodes` construct provides a way to define the set of operations or instructions that belong to a particular slot.

Example:

The following example defines a single 32-bit wide format. The format consists of 2 slots called `slot_a` and `slot_b`.

```
format f32 32 {slot_a, slot_b}
```

The following TIE code shows the assignment of the instructions to the slots.

```
slot_opcodes slot_a {ADDACC, SUBT}
slot_opcodes slot_b {SUBT, ADD16}
```

Notice that the instruction SUBT is now implemented twice in hardware since it is assigned to both slots. In C/C++ the compiler automatically executes the instructions assigned to different slots in 1 single clock cycle.

1 cycle:

```
diff = SUBT(5, 12);
ADDACC(acc, a);
```

1 cycle:

```
diff_1 = SUBT(5, 12);
diff_2 = SUBT(1, 7);
```

2 cycles:

```
ADDACC(acc_1, a);
ADDACC(acc_2, b);
```

TIE Built-in Modules

TIE provides a set of commonly used operators as built-in modules invoked in a way similar to function calls. The modules have been pre-optimized for improved timing and reduced area. The use of TIE modules generally results in TIE with more optimal timing and area compared to TIE described using standard TIE operators. The following table

gives a short overview of all built-in modules. Please consider Table 4-2 in the *TIE User's Guide* and Section 29 of the *TIE Reference Manual* for more details.

Built-in module example	**Definition**
`sum = TIEadd(a, b, cin)`	sum = a + b + cin
`sum = TIEaddn(A0, A1, ..., An-1)`	sum = A0 + A1 + ... + An-1
`{carry, sum} = TIEcsa(a, b, c)`	carry = a & b | a & c | b & c sum = a ^ b ^ c
`{lt, le, eq, ge, gt} =` `TIEcmp(a, b, signed)`	lt = (a < b) le = (a <= b) eq = (a == b) ge = (a >= b) gt = (a > b)
`o = TIElzc(a)`	o = a[w-1] ? 0 : a[w-2] ? 1 : ... a[0] ? (w-1) : w (w is the computed width of a)
`o = TIEmac(a, b, c, signed, negate)`	o = negate ? c - a * b : c + a * b
`prod = TIEmul(a, b, signed)`	prod = a * b
`{p0, p1} =` `TIEmulpp(a, b, signed, negate)`	p0 + p1 = negate ? - a * b : a * b
`o = TIEmux(s, D0, D1, ..., Dn-1)`	o = s == 0 ? D0 : s == 1 ? D1 : ... : s == n-2 ? Dn-2 : Dn-1
`o = TIEpsel(S0, D0, ..., Sn-1, Dn-1)`	o = S0 ? D0 : S1 ? D1 : ... : Sn-1 ? Dn-1 : 0
`o = TIEsel(S0, D0, ..., Sn-1, Dn-1)`	o = (size{S0} & D0) | (size{S1} & D1) |... (size{Sn-1} & Dn-1);

Function

A TIE `function` is similar to a function in C or Verilog. It encapsulates some computation, which can then be used in different places in a TIE description.

Example:

The following example defines a function `addsub` that either computes the sum or the difference of 2 8-bit values. The operation `ADD8X4` performs 4 8-bit additions in parallel.

```
function [7:0] addsub ([7:0] a, [7:0] b, add) {
     wire [7:0] tmp = add ? b : ~b;
     wire cin = add ? 0 : 1;
     assign addsub = TIEadd(a, tmp, cin);
}
operation ADD8X4 (out AR sum, in AR in0, in AR in1) {} {
     wire [7:0] t0 = addsub(in0[ 7: 0], in1[ 7: 0], 1'b1);
     wire [7:0] t1 = addsub(in0[15: 8], in1[15: 8], 1'b1);
     wire [7:0] t2 = addsub(in0[23:16], in1[23:16], 1'b1);
     wire [7:0] t3 = addsub(in0[31:24], in1[31:24], 1'b1);
     assign sum = {t3, t2, t1, t0};
}
```

Every reference to a function instantiates a new copy in hardware of the logic implemented by the function. By adding the keyword `shared` to the function, only a single copy is instantiated in hardware. This allows to share the resources and to reduce logic area. Notice that, sharing the function `addsub` in the example above would cause a resource interlock. As a consequence, the operation `ADD8X4` would take 4 cycles instead of 1 cycle to finish.