Contents

• Purpose and place in the tool chain
• TIM, hive processor description language
• HSD, hive system description language
Design Meth. II: Design-space Exploration

- Processor Design Entry
- Pre-Configured IP / blocks
- Processor Generator
  - HDL code
  - standard EDA tools
  - netlist layout
- Processor instance
- TIM
  - SYNTHESIS & BACK-END
  - ARCHITECTURE SPACE EXPLORATION
  - HiveCC
    - Assembly
    - Assembler & Linker
    - Simulator
    - Execution Trace
- Application Programs (C)
  - minutes to hours
Fully automated flow from system description to RTL

- HiveLogic Pre-Configured IP / blocks
- Custom IP
- Cell Descriptions
- System Description
- Application Programs (C)

- Verification IP
- GeneSys
- TIM+HSD
- HiveCC
- Core Browser
- Assembler & Linker
- Assembly
- Multi-core System Simulator

- Testbench Generator
- HDL code
- standard EDA tools
- System instance
- Cell instance

- HiveLogic
- Processor Descriptions
- Processor Descriptions
- Processor Descriptions
- Cell Descriptions
- Processor Descriptions
Processor Description in TIM
Processor (sub-system) template
TIM is only about CELL

Processor:
- Sequencer
- Instr. decoder
- Loopcache

Register file

Issue slot

Function unit

Data routing network

Memory device

Arbiter

Address map
TIM Hierarchy

Core

- Processor
  - Cell
    - Register File
    - Issue Slot
    - Logical Memory
    - Cluster
      - Add_map, slave address mapping

Core IO

- Logical Memory
  - Add_map, core address mapping
  - Device, storage and io
    - ram
    - FIFO
    - Master Interface
Core & Core IO

Core

- Core IO ports
- Sequencer ports

Core IO

- Stream ports
- Master ports
- Slave ports
TIM building block

• The principle of the TIM language is to define, instantiate, parameterize, and connect building blocks (BB),

• The syntax of a building block is generic and allows:
  ▪ Declaring a new building type
  ▪ Specifying its interface, input and output ports
  ▪ Being instantiated multiple times
  ▪ Declaring its inner building blocks (hierarchy)
  ▪ Setting its properties
  ▪ Setting its parameters
TIM building block syntax

BB_type  BB_name  ( PortType in0, PortType in1, ...)
->  ( PortType out0, PortType out1, ...)

{
  body
  out0 = ...;
  out1 = ...;
}

• <BB_type>, is either a reserved type from the TIM hierarchy (IS, FU, RF ...), or a user defined building block name (aka. Derivation)

• <BB_name> is user choice

• <Body> can contain properties, parameter initializations, instantiation connection or can be empty
TIM building block syntax
TIM building block syntax

BB_type  cera  ( PortType in0, PortType in1 )
          -> ( PortType out0, PortType out1 )
{
    // instantiation of building block ladybug, and its input connections
    ladybug_BB_type ladybug ( in0, in1 );

    // output connections
    out0 = ladybug.output0 ;
    out1 = ladybug.output1 ;
};
TIM building block syntax

BB_type ladybug ( PortType input0, PortType input1 )
   -> ( PortType output0, PortType output1 )
{
   // may contain instantiation of other units
   ant ant1 (input0, input1);
   ant ant2 (input0);
   bee bee1 (input0, input1);

   output0 = { ant1.output0, ant2.output0, bee1.output0 };
   output1 = bee1.output1;
};
TIM building block syntax
Properties

• Each building blocks of the TIM hierarchy can set different properties

• Example #1

Cell pearl (inputs) -> (outputs)

{ 
  CharBits := 8;
  ShortSize := 2;
  DefaultMem := bp.dmem;
  ... 
};

• Hierarchy is expressed with “.”

• Example #2

Bus Bus32 (Port32 in) -> (Port32 out)

{ 
  Width := 32;
  ...
};

• Properties use “:=”
• Connections use “=”
Parameters

• Declaration example of a parameterized building block:

```c
BB_type my_block < signed p0, signed p1 > ( inputs ) -> ( outputs )
{
    signed new_param := p0 + p1; // parameter expression, supports ...
};
```

• Instantiating a parameterized building block:

```c
BB_type my_other_block (inputs) -> (outputs)
{
    signed param0 := 2; signed param1 := 2;
    my_block bug_with_params < a, b > ( inputs );
};
```
Parameters and properties

• Parameters are used in different places

BB_type my_BlockNxB < signed nway, signed elembits > /* no inputs */

    -> (Port <width, elembits > output)

{

    signed width := nway * elembits ; // to compute new parameters, +, -, *, /, %, <<, >>

    Property_0 := nway;            // to set properties

    Property_1:= elembits;        // parameter expression

    bb_type inst0 < nway, width> ( inst1.o0) ; // instantiation

};
Machine Description

- Processor
- Cell
- Bus
- Cluster
- Issue slot (IS)
- Functional unit (FU)
- Register File (RF)
- Operation (OP)
Processor

- TIM keyword: Processor
- Specification of input and output ports of processor: master IF, slave IF, fifo IF
- Instantiation of cell
- Specification of inter-Cell connectivity (currently 1 cell only: processor I/O = cell I/O)
- Inter-Processors connectivity is done with “Hive System Description” (HSD)
Processor: TIM code

Processor pearl_coral_processor

( mmioW_DTL<intWidth> sl_ip, fifoW_DTL<bpFifoWidth> st_ip0, st_ip1)
-> ( mmioW_DTL<intWidth> sl_op, fifoW_DTL<bpFifoWidth> st_op0, st_op1)

{
/* parameters that hold throughout the whole cell */
signed intWidth := 32;
signed llWidth := 40;
signed bpFifoWidth := 32;

/* instantiation of Cell */
pearl_coral_cell pearl_coral <intWidth, llWidth, bpFifoWidth> (sl_ip, st_ip0, st_ip1);

/* list of outputs of this building block (Processor) */
sl_op = pearl_coral.sl_op;
st_op0 = pearl_coral.st_op0;
st_op1 = pearl_coral.st_op1;
}

Cell

- TIM keyword: Cell

- Specification of input and output ports of cell
  (currently 1 cell only: processor I/O = cell I/O)

- Instantiation of:
  - clusters, Issue slots, register files, logical memories,
  - sequencers, slave address mapping, logical memories

- Interfaces
Cell: tim code (1)

Cell pearl_coral_cell < signed intWidth, signed llWidth, signed bpFifoWidth >

( mmioW_DTL<intWidth> sl_ip, fifoW_DTL<bpFifoWidth> st_ip0, st_ip1 )
-> ( mmioW_DTL<intWidth> sl_op, fifoW_DTL<bpFifoWidth> st_op0, st_op1 )

{

/* C-types size properties */
CharBits := 8; ShortSize := 2; IntSize := intWidth/8;
LongSize := 4; LongLongSize := llWidth/8;

/* Default memory */
DefaultMem := bp.dmem;

/* Stack pointer and function call return pointer */
SP_rf := bp.rf2; SP_idx := 0; RP_rf := bp.rf2; RP_idx := 1;

/* optional: function call parameter passing registers */
/* parameters for the cluster pearl */
signed bpRF1cap := 16;
signed bpRF2cap := 16;
signed bpMemCap := 16384;

/* instantiation of cluster bpse_pearl with name bp */
bpse_pearl bp < intWidth, bpRF1cap, bpRF2cap, bpMemCap, more parameters... >
    ( coral1.op, sl_ip, sl_ip, sl_ip, st_ip0, st_ip1 );
/* instantiation of cluster cpse_coral with name coral1 */
cpse_coral coral1 < parameters... >
    ( bp.op, sl_ip );

/* list of outputs of this building block (Cell), only one output */
sl_op = { bp.sl_op_config, bp.sl_op_pmem, bp.sl_op_dmem, coral1.sl_op };
And so on

- TIM keyword: Bus, Cluster, IS, RF, FU, OP, SEM and Port

- Derivation mechanism:

  FU BASE_ALU (Port a, b) -> (Port r)
  {
  <operations>
  }

  BASE_ALU BASE_ALU_16 (Port16 a, b) -> (Port16 r)
  {
  <operations>
  }

- This mechanism can be applied to all building blocks
• Scalar multiplication example

```c
OP std_mul (Signed a, b) -> (signed r) {
  SEM r (a,b) = { r = (sr) a * (sr) b ;}

  <DOC>
  <SHORT> Signed multiplication </SHORT>
  <SEM> r = a * b </SEM>
  <DESCRP> this operation returns the signed product of the
           arguments a and b. < \DESCRP>
  <\DOC>;
}
```

• Where sr is cast defined as: signed<Width:=(Width(R))>

• TIM has its own type checker
Operation semantics, example #2

• Vector multiplication example

\[
\text{OP} \quad \text{vec\_mul} <\text{signed nway}> (\text{svecN }<\text{nway}> A, B) \rightarrow (\text{svecN }<\text{nway}> R) \quad \{ \\
\quad \text{SEM} \quad R <\text{nway}> (A, B) = \{ \quad /* \text{loop over each vector element} */ \\
\quad \quad \text{for } i \in [(\text{nway}-1), 0] \\
\quad \quad \quad \{ \\
\quad \quad \quad \quad r[i] = (\text{sRi}) \text{ std\_mul.r}(A[i], B[i]); \\
\quad \quad \quad \}; \\
\quad \quad <\DOC> \quad \ldots \quad </DOC> \\
\quad \}; \\
\]

• Where sRi is cast defined as: signed<Width:=(Width(R), Elems(nway)> 

• TIM also performs function inlining, constant folding ... like any compiler!
Core IO, Logical Memory (ram)

NB: slave routing network (SRN) is at Cell level, not inside an Logical_memory
Logical Memory (ram)

Device dev_dmem_arb <signed width, signed memCap>
   ( mmioW_p_in<width> wp )
-\>
   ( mmioW<width> rp )
{
   Width := width;
   Capacity := memCap;
   Kind := ram;
   Latency := 1;
}

Logical_memory lm_2mem_sl <signed width, signed memCap>
   ( mmioW_p_out<width> core_ip, sl_ip0, sl_ip1 )
-\>
   ( mmioW<width> core_op, sl_op0, sl_op1 )
{
   dev_dmem_arb mem0 <width, memCap> ({ core_ip, sl_ip0 });
   dev_dmem_arb mem1 <width, memCap> ({ core_ip, sl_ip1 });

   core_op = { mem0.rp, mem1.rp };
   sl_op0 = mem0.rp;
   sl_op1 = mem1.rp;
};
Logical Memory (fifo)
Logical Memory (fifo)

Device fifo_2way <signed fifoWidth, signed fifoCap>
   (fifoW<fifoWidth> ip0, ip1) -> (fifoW<fifoWidth> op0, op1)
{
    Width := fifoWidth;
    Capacity := fifoCap; // minimum is 2. In words of Width bits
    Kind := fifo;
    Latency := 0;
};

Logical_memory lm_1fifo <signed ffWidth, signed ffCap>
   (fifoW<ffWidth> core_ip, st_ip) -> (fifoW<ffWidth> core_op, st_op )
{
    fifo_2way     fifo0 <ffWidth, ffCap> ( core_ip, st_ip );

    // note: at these output assignments the "wiring is crossed" deliberately:
    st_op         = fifo0.op0;
    core_op       = fifo0.op1;
};
Logical Memory (program mem)

address mapping is in Cell level
Logical Memory (stat&ctrl)

address mapping is at Cell level
Address Mapping

• Syntax

Add_map (inputs) -> (outputs)
{
    output_port_id = input_port_id [ <start_address> , .. , <end_address> ];
};

• Example:

Add_map mrn_data_mem (MMIO32 ip0) -> (MMIO32 op0, MMIO32 op1)
{
    op0 = ip0 [0x00000, .., 0x03FFF];
    op1 = ip0 [0x04000, .., 0x07FFF];
};
Device

• Syntax:

Device ( input ) -> ( output )

{
    Width := expr;
    Capacity := expr;
    Kind := [ram | rf_ram | master_int | stat_ctrl | prg_mem | fifo];
    Latency := expr;

    /* for master interface only: */
    BaseAddrs := expr;
    MaxBurstSize := expr;

};
TIM file structure

• To get some structure in the files for a core, a general partition of the TIM source files could look like:

  corename.tim  processor and cell
  corename_is.tim  issue slots
  corename_fu.tim  (derived) functional units (if not from lib)
  corename_lm.tim  logical memories

• If the core is partitioned with clusters the TIM source code for each cluster could be partitioned the same way
PBB library: overview

• PBB: Processor Building Blocks

• Contains coarse to fine grain granularity building blocks described in TIM

• Use coarse blocks for quick starting point

• Use fine grain blocks for more control/optimization

• The building blocks become available using #include in your core files

• They are self-contained: the blocks themselves include the lower level blocks they need.
Hands on
Hands on TIM

• Lab : Add another issue slot in the Hive Processor
  • use issue slot s2 as the base for the new issue slot
  • remove the XLSU FU in the new issue slot
  • connect the new issue slot similarly as s2
  • might need to duplicate more things ...

• Compiling command :
  • $HIVEBIN/../libexec/tim -I. scalar2400_wider.tim
  • NB: Ignore warnings

• Visual inspection :
  • $HIVEBIN/corebrowser
Hive System Description or HSD

- Reuse the syntax of TIM language
- Use to describe a system including
  - Host
  - Busses (or fabric)
  - Hive Processors
  - FIFO(s)
  - Devices
  - Custom devices
- Input to retarget the simulator
- Generate the address map description file (RDL format) for documentation
One difference in the language

• The address map for the busses (fabric) is enhanced as follow:

Device SystemBus (mmio_port <32,32> ip0, ip1)

    -> (mmio_port <32,32> op0, op1)

{
    op0.Address := 0x0;
    op0.Capacity := 0x5000;
    op1.Address := 0x5000;
    op1.Capacity := 0x1000;
}

NB: parameters can be set directly using constant but it is not recommended for portability
Hands on
Hands on HSD

- Lab 1: Insert a hive processor in a system
- Lab 2: Create the address map for the Systembus (fabric)

- NB: Compiling command
  - $HIVEBIN/../libexec/hsd <FILE>.hsd
HSD Lab #1

• Add an instance of the hive processor scalar_2400_processor in the system

• Interface connection:
  • No soft reset connected, use keyword “None”
  • its two slave ports come from the System bus, sl_ip0 and sl_ip1
  • its streaming ports are connect to the following FiifoAdapter bocks :
    • scalar_processor_2400_fifo_fifo[0 to 10]
  • its master interface output port connects to the CIOConverter imt_op block
    • its output slave port connects directly to the host

• Its master interface must have a base address set to : 0x380000
• Its instruction cache must have a base address set to : 0x380000
HSD Lab #2

- Define the address map for the System bus as follow
  - Capacity for port sl_ispstatctrl = 0x3F
  - Capacity for port sl_ipdem = 0x3FFF
  - Capacity for port sl_ippmem = 0x3_FFFF
  - Capacity for port sl_ipvamem = 0x1_FFFF
  - Capacity for port sl_isphist = 0xFFF
  - Capacity for port simd_ipdma = 0x1FF_FFFF
  - All the slave ports connected to the FifoAdapters have a Capacity = 0x10
    - Capacity for port gp_reg = 0x10
    - Capacity for port external_memory = 0x40000