TUNI
TNO-FEL

Architecture Definition File

Processor Architecture Definition File Format for a New TTA Design Framework

Authors: *A. Cilio (Tampere University)*
H.J.M. Schot (TNO-FEL)
J.A.A.J. Janssen (TNO-FEL)
Pekka Jääskeläinen *(Tampere University)*
Kati Tervo (Tampere University)

Unit of Computing Sciences

TNO Physics and Electronics Laboratory

Summary

This document gives the complete reference specification of the Architecture Definition File Format (ADFF) for a TTA codesign framework. This format is used to specify the Architecture Definition File (ADF) used to define a target TTA processor's architecture. The work presented in this document is a joint effort of Tampere University (Finland) and TNO-FEL (The Netherlands).

Document History

Version	*Date*	*Author*	*Comment*
0.1	07/02/03	A. Cilio	First draft of the document.
0.2	14/02/03	J.A.A.J. Janssen	Comments after review
0.3	21/02/03	H.J.M. Schot	Comments after review
0.4	14/03/03	A. Cilio	Complete review of comments in v0.2-0.3 and decisions taken during conference call on 10.03.03.
0.5	24/03/03	A. Cilio	Partial revision after conference calls on 17.03 and 20.03.
0.6	17/04/03	A. Cilio	Complete revision after conference calls up to 24.03 and comments sent by email.
0.7	06/05/03	H.J.M. Schot	Additions to MDF implementation.
0.7b	07/05/03	H.J.M. Schot	XML encoding definition.
0.7c	07/05/03	A. Cilio	Completed draft of MDF specification in XML format.
0.7d			
0.7e	08/05/03	A. Cilio	Comments to MDF encoding appendix.
0.7f	12/05/03	H.J.M. Schot	Updated comments to MDF encoding appendix.
0.7g	15/05/03	A. Cilio	Revised MDF implementation. Added section on validation. Revised MDF encoding appendix.
0.7h	20/05/03	H.J.M. Schot	Simplified pipelining model Function Units
0.7i	22/05/03	A. Cilio	Added new guard support declaration. Modified and commented MDF implementation.
0.7m	03/06/03	A. Cilio	Modifications to port declaration syntax in long immediate block (mostly undoing a change in 0.7i).
0.8	27/06/03	J.A.A.J. Janssen	Finalizing document.
0.9	16/07/03	A. Cilio	Corrections and comments. RF type, socket declaration.
0.10	31/07/03	J.A.A.J. Janssen	Comments and corrections to last modifications.
0.11	01/08/03	A. Cilio	Added segmented and clustered transport networks.
0.12	01/10/03	A. Cilio	Added latency to Immediate Unit. Added read/write acces constraint parameters to RF definitions. Relaxed Bridge/Bus connection limits. Defined empty instruction template. Redefined completely the operation input/output specifications, FU Port properties, Execution Pipeline specifications. Reintroduced transport pipeline declaration. Many other minor corrections.
0.13	22/10/03	A. Cilio	Modifications to address space declaration and references. Revised description of pipeline model. Corrected 0 start-cycle.
0.14	7/11/03	A. Cilio	Corrections according to comments of J. Heikkinen and V.

Version	*Date*	*Author*	*Comment*
			Guzma. Changes to pipeline declaration, added example of zero-latency operations.
0.15	13/11/03	A. Cilio	Redefined segmented bus and clustering specifications.
0.16	02/12/03	A. Cilio	Reorganised validity constraints. Added width to GCU ports.
1.0	28/01/04	A. Cilio	Added bus XML constraint (mandatory segment element, at most one unconditional guard per bus).
1.0.1	12/02/04	A. Cilio	Clarified description of the pipeline resource model.
1.0.2	11/03/04	A. Cilio	Added dependency diagram between top-level elements.
1.0.3	19/03/04	A. Cilio	Revision after meeting at NRC, Helsinki, on 17.03.
1.0.4	26/04/04	A. Cilio	Complete revision of validity constraints. Segment element optional again. Address space always mandatory, but its word width can be empty in control unit.
1.0.5	06/05/04	A. Cilio	Important correction in a complex socket XML constraint.
1.0.6	29/06/04	A. Cilio	Added socket constraint about non-straddling unit. Added definition of Unique Name Attribute. Minor corrections.
1.0.7	03/09/04	A. Cilio	Corrected and added range constraints of certain integer parameters. Binding of operation outputs to FU ports not mandatory. Relaxed canonicity constraint. Minor corrections.
1.1	10/10/04	A. Cilio	Format revision. Renamed ‘MDF’ to ‘ADF’. Changed pipeline declaration and GCU ports. Removed address space interleaving factor. Minor corrections throughout. Rejected distinction of actual ‘trap’ operation from simulator hook.
1.1.1	12/10/04	A. Cilio	Revision after review by L. Laasonen.
1.1.2	08/01/05	A. Cilio	Clarified and relaxed constraints of instruction template.
1.2	17/01/05	A. Cilio	Restricted character set of Unique Name Attribute. Revised GCU. Removed *control* element, added *port* element. Added constraint to *bind* element of hardware operations.
1.3	26/01/05	A. Cilio	Revision after review by L. Laasonen and A. Oksman. Introduced *special-port* in GCU.
1.3.1	28/01/05	A. Cilio	Element *return-address* mandatory and nillable.
1.3.2	21/02/05	A. Cilio	Legal name of operations restricted to lower case.
1.4	05/05/05	A. Cilio	Revised guard support, added “always-false” expression.
1.5	19/05/05	A. Cilio	Added *immediate-slot* element. Clarified that the choice of immediate register in instruction template tag is orthogonal.
1.5.1	11/08/05	A. Cilio	Clarifications regarding bit width adjustment.
1.5.2	05/01/06	P. Jääskeläinen	Allow underscore in operation names.
1.5.3	09/02/06	L. Laasonen	Minor clarification about how sockets are connected to buses.
1.6	13/04/06	A. Cilio	Added *guard-latency* elements.
1.6.1	25/04/06	A. Cilio	Removed local guard latency. Global latency can be zero.

Version	*Date*	*Author*	*Comment*
1.6.2	11/05/06	L. Laasonen	Removed the constraints of max reads and max rw of Register File. Added local guard latency for Register File.
1.6.3	30/05/06	L. Laasonen	Removed XML validity constraints concerning socket-segment and socket-port connections if there are bridges in the machine.
1.6.4	06/09/07	P.Jääskeläinen	The 'transport-stages' property of control unit declaration replaced with more generic 'delay-slots' property (minimum value zero). The guard-latency property of RF (and IU) can be either 0 or 1 only. Added 'max-writes', removed 'max-rw'. Immediate Unit: Unified the with RF. Removed the 'cycle' property (latency fixed to 1). 'max-writes' is always 1 and its not sanity checked against the count of write ports attached to it. Clarifications to the Immediate Support Chapter. RF ports cannot be reused by multiple buses, thus the same port cannot be read by multiple moves at the same cycle. This differs from FU output ports in which case the same port can be read by any number of times through different connected buses.
1.7	17/10/07	P.Jääskeläinen	Upped the version number. From now on, changes that do not modify the actual format should increment only the third number of the version number. Thus, the second number denotes the actual ADF format version between which compatibility is not guaranteed. This is mainly because XML schema supports only one level of subversions. This version of ADF is the one used in TCE v1.0.
1.7.1	08/03/10	P.Jääskeläinen	Added the reasoning behind leaving the upper bits undefined when reading from a narrower source to a wider to the socket section.
1.7.2	22/12/14	P.Jääskeläinen	Reversed the earlier constraint that RF ports cannot be reused by multiple buses. It is now officially possible to read the same register to multiple buses through the same RF port. If referring to different indices, the result is undefined.
1.8.0	17/07/19	K.Tervo	Added little-endian tag as a top-level specifier. ADFs without the specifier are assumed to be big-endian.
1.9.0	17/07/19	K.Tervo	Changed guard evaluation logic in Transport Buses. Now, the result of the guard evaluation is simply the least significat bit, rather than a reduction OR of the value.

Table of Contents

1. Introduction

Microprocessors used in embedded systems often have specific requirements like low cost, high performance and low power consumption. COTS microprocessors can not always fulfill all requirements. A templated processor architecture, which can be tuned for a certain application (domain), offers a solution. The Transport Triggered Architecture (TTA) offers such templated processor architecture. TTAs combine flexibility, modularity, and scalability and can be customised for use in a wide variety of products, including image processing, telecommunications and consumer electronics products.
The TTA concept is targeted for use in Application Specific Processors, or ASPs. A TTA can be considered as a collection of Function units (FU's) register files (RF's), Immediate units (IU's), transport buses, and sockets. FU's perform operations, RF's provide temporary fast accessible storage, the network of buses performs data transports between the FU's and RF's, and sockets interface FU's and RF's to transport buses. The designer can add as many types of FU's and RF's as required for the application. When required, new application specific operations can be added to speed up execution time. TTAs are programmed by data transports: a program is a collection of *moves* that specify how data must be moved between FU's and RF's. As a side effect of data transports, FU's execute operations. A TTA can perform multiple moves, and thus execute multiple operations, in parallel. The interconnection network may be fully connected –in which case every socket is connected to all move buses– or partially connected. A single bus can carry out one or more data transports in a single cycle. In the latter case, the bus is subdivided into independently programmable *segments*. TTAs can be *clustered*. In a clustered TTA, each bus belongs to one cluster and can perform only data transports between RF's and FU's of the same cluster in one cycle. Inter-cluster data transports are enabled by *bridges*, which connect busses that belong to different clusters. To facilitate conditional execution, control flow changing operations and predicated execution is supported. Figure 1 shows the general structure of a TTA.

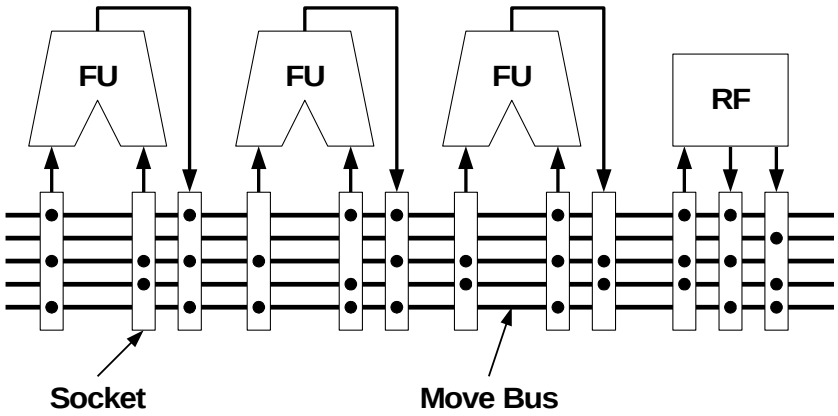

Figure 1. General structure of a TTA.

An instance of a TTA is specified with an Architecture Definition File. Such a file describes the type and amount of resources (FU's, RF's, buses, sockets, etc.) and their interconnections. This document gives the complete reference specification of the Architecture Definition File Format (ADFF) on which ADF's are based.
This document is split in two parts. In part 1 the requirements for the architecture

definition format are identified. Based on these requirements an XML specification is proposed. This XML specification is presented in part 2.

Part 1: Requirements

In this part the requirements of the ADF are specified. For each component of the processor the relevant parameters are identified. In addition, constraints of the architecture template are given.

2. Endianess

An architecture may be either big-endian or little-endian. The endianess determines the default data layout used for the architecture and, by extension, the load and store operations used to access data. In a little-endian architecture, an integer word that is composed of multiple minimum addressable units of an address space is arranged in memory with the least significant unit in the lowest address and the most significant unit in the highest address, while a big-endian architecture arranges them in reverse.

3. Transport Buses

The function of transport buses is to transport (or move) data between processor units (Function Units, Register Files, Immediate Units, Global Control).

3.1.Parameters

Transport bus parameters	
TB-1	Bit width of the bus data path.
TB-2	Support type for predicated execution of transports.
TB-3	Segmentation.
TB-4	Sockets connected to the bus.

3.2.Remarks

TB-1: The bit width of the bus data path must be a positive integer number.

TB-2: The support type for predicated execution is specified independently for each transport bus. Two types of predicated execution are supported:

1. *Unconditional execution*
   A data transport is unconditionally executed when its execution does not depend on machine state. There are two situations where unconditional execution can take place.
   1. When predication is not supported at all on a bus. In this case, it is implied that a transport is always carried out to completion.
   2. When predication is supported on a bus, and a special guard expression that represents the "always true" condition is selected for a move.[1] In this case, another type of unconditional guard, which represents the "always false" condition, can be supported by the same bus.[2]
2. *Conditional execution*
   The execution condition is expressed in the form of a guard expression. Guard expression specifications are made of the following elements:
   1. *Boolean register terms*. These terms refer to a register in a register file and represent its contents (treated as a Boolean value);
   2. *Function Units result terms*. These terms refer to the result port of a function unit and make it possible to forward the result of an operation (treated as a Boolean value –typically, a comparison result) to the predicate evaluation logic;
   3. *An optional binary operator* that applies to two guard terms and computes a logical operation. An operator combines the Boolean value of two terms into one Boolean value. Two operators are admitted:
      a. Logical intersection (and);
      b. Logical union (or);

[1] For practical reasons of programmability, it is recommendable that any bus that supports one or more guard expressions should support also the special "always true" Boolean expression.
[2] For efficient implementation of decoding logic, it is reasonable to assume that the "always-false" guard expression is supported when an "always-true" expression is.

4. *An optional unary operator* that applies to a single guard term (and so has precedence over binary operators) and inverts its value.

***TB-3*:** A bus may be subdivided into *segments.* A data transport on a segmented bus can span one or more segments. A segmented bus can perform several transports in a single cycle, as long as the sequences of segments used by the transports do not overlap. Each segment is fully programmable like an independent bus.

Evaluation of a guard term. When a GPR or an output from a function unit is evaluated as term of a guard expression, only the least significant bit is considered. If the bit is 1, then the guard term evaluates to true, otherwise, it evaluates to false.

3.3.Constraints

1. At least one transport bus must support the minimum level of predicate execution support needed to make program control flow possible. The minimum level of predicate execution support is defined as:
    a. One register term or Function Unit result term;
    b. Support for the logical negation unary operator.
2. The segments of a bus must form one unidirectional chain whereby a segment may be driven by another segment and may drive a third segment.

4. Sockets

Sockets are programmable connections. Each socket connects one or more sources or destinations with one or more transport buses. Sources and destinations may be input/output ports of various types of components: Function Units, Immediate Units, or Register Files. In the rest of this document, any source or destination will be generically referred to as *port*.

4.1.Parameters

Socket parameters	
S-1	Bus segments connected to the socket.
S-2	Direction of the socket (input/output).

4.2.Remarks

S-s: Sockets are either input or output. Bidirectional sockets are not supported.

Port-socket connection**:** A socket can read from or write to several ports. The ports connected to a socket can belong to different components (Function Units, Register Files, Immediate Unit and Global Control). Thus, several ports can share a socket.

Socket bit width**:** The bit width is not a parameter of a socket, but rather of individual port-bus connections. The bit width of a connection depends on the bus and the port bit width. When the bit widths are not equal, four cases can occur:

1. Input socket: width(bus) > width(destination).
2. Input socket: width(bus) < width(destination).
3. Output socket: width(bus) > width(source).
4. Output socket: width(bus) < width(source).

In cases (1) and (4), a number of incoming data lines are simply not connected through to, respectively, the destination port or the bus. This type of connections can be used on condition that the data is transported without losing significant bits.

Case (2) is the most problematic. An extension of the data coming from the bus must take place. The bits undergoing extension are by definition *undefined*. No assumption should be made about the value of the extended bits; programs should not depend on those bits.

Also in case (3), the values on the bus lines not connected to the output port are undefined.

In all cases, the bits involved in extension or reduction are those that occupy the most significant bit positions, i.e., the least significant bits are aligned.

5. Bridges

Bridges are optional components. By specifying bridges, it is possible to describe *clustered* architectures.

A bridge can be programmed to write the value currently on a bus onto another bus. A bridge takes one cycle to drive the output bus with the value read from the input bus. Two busses connected by a bridge become segments of a longer transport path. A path is thus a chain of two or more busses.

5.1.Parameters

Bridge parameters	
BR-1	Input bus connected to the bridge.
BR-2	Output bus connected to the bridge.

5.2.Remarks

***BR-1*:** A bridge is unidirectional and reads from one and only one input bus. In turn, an input bus can be connected to the output of another bridge.

***BR-2*:** A bridge is unidirectional and writes onto one and only one output bus. In turn, an output bus can be connected to the input of another bridge.

The equivalent of a bidirectional bridge is modeled by two independent bridges attached to the same bus pair, but with opposite direction.

5.3.Constraints

1. A bus chain cannot form a cycle, that is, there is never a path such that it is possible for the output bus of a bridge to drive its value onto the input bus.
2. A bus can be connected to no more than two other busses by at most four bridges (two output and two input bridges).
3. No socket can write to (or read from) two busses of the same bus chain.
4. No socket can write to (or read from) two busses B_1, B_2 if there exists another socket that writes to (or reads from) B_1 and also writes to (or reads from) a bus B_3 that is joined to B_2 through more than one bridge.
5. Given a socket connecting two busses B_1, B_2, no other socket can write to or read from two other busses B_3, B_4 of the same bus chains to which, respectively B_1 and B_2 belong unless B_3, B_4 are in the same relative position with respect to B_1 and B_2.

6. Function Units

A Function Unit (FU) implements one or more operations of the target architecture's instruction set. An FU communicates with the rest of the processor through a number of input/output ports.

6.1.Parameters

Function Unit parameters	
FU-1	Supported operation set.
FU-2	Input and output ports (operands and results of operations).
FU-3	Bit width of input and output ports.
FU-4	Binding between sockets and ports.
FU-5	Memory address space accessed by the FU.

6.2.Remarks

***FU-1*:** The operations implemented in a Function Unit are referred to as *Hardware Operations*, described in Section 7.

***FU-3*:** For each port the bit width is specified independently.

***FU-4*:** For each port the connected socket is independently specified. Several ports (even belonging to different units) can share the same socket. Thus, sockets are not necessarily tied to a single port.

***FU-5*:** If an FU contains Hardware Operations that can access memory (see Section 7), then the FU declaration must specify the address space (described in Section 9.3) that is accessed by the FU.

6.3.Constraints

1. Each FU input (output) port is connected to only one input (output) socket.
2. A FU port can be connected to two sockets, one writing to and one reading from busses. In this case, the port is bidirectional.
3. If one of the containing Hardware Operations accesses memory, then the FU declaration must contain an address space specification.
4. A FU can access only one memory address space.

7. Hardware Operations

Hardware Operations represent the physical implementation of an operation onto a specific FU. As such, Hardware Operations are part of the FU declaration, because their properties are fully defined by a combination of their individual properties and the FU properties.

7.1.Parameters

Hardware Operation parameters	
HO-1	Parent Function Unit.
HO-2	Name of the base operation.
HO-3	Binding between operands/results of the operation and FU ports.
HO-4	Pipeline modelling specification.

7.2.Remarks

***HO-1*:** Each Hardware Operation belongs to one and only one Function Unit.

***HO-2*:** The name of the operation uniquely identifies the operations performed, its properties and the number of inputs and outputs. This information is collected in a separate database [] and is out of the scope of this document.

***HO-3*:** A Hardware Operation must specify the binding between the inputs and outputs of the base operation and the ports of the Function Unit. This is an additional degree of freedom: it is possible to specify separate FU Ports for each input and output of different operations.

***HO-4*:** A Hardware Operation must contain the specification of its execution pipeline.

Pipeline Model. Pipelines are defined using the concept of *resource*. Usually, a resource represents a piece of hardware that implements part of the pipeline, but in abstract, any constraint in how a pipeline is used can be modeled by means of resources. A resource is identified by an arbitrary name string that is unique within the Function Unit scope. A pipeline declaration consists of a finite sequence of *resource usage* elements. Each element stands for one or more cycles in which pipeline resources are claimed, and consists of one or more resource identifiers combined with the following two syntactic elements: (1) union of resource identifiers (= two otherwise independent resources are claimed simultaneously) and (2) repetition of resource usage (= the resource usage extends for a given number of cycles).

A few examples should clarify the use of this model. Unions are indicated by “+”; repetitions by “^”, and sequence element separators by blank spaces. The syntax used is arbitrary. The implied scope for a resource usage element is the entire FU.

Example 1: Unpipelined, 5-cycle (integer) multiply

```
s1^5
```

Example 2: Fully pipelined, 4-stage load

```
s1 s2 s3 s4
```

Example 3: Floating-point divide operation with a 2-cycle unpipelined unpack and a 2-cycle pipelined pack phase and a 10-cycle unpipelined computation phase, where one cycle of the computation phase overlaps with the unpack phase

```
s1 s1+s2 s2^9 s3 s4
```

This model of pipeline declaration provides a mechanism to specify whether two operations compete for the same resources. If two Hardware Operations in a Function Unit have exactly the same execution pipeline declaration, then the same pipeline implements both operations. If the pipeline declarations contain a reference to the same resource usage element, then the two operation pipelines have a block of hardware in common. The semantic of a resource usage element is comparable to that of the bits in reservation vectors often used by instruction schedulers.

Independent operation pipelines. The pipeline is a property of individual operation implementations, not FU's. It is possible to have an FU with multiple, independent pipelines. Since independent physical ports can be associated to operand/results of operations, operations on the same FU may use completely independent hardware resources. An FU with such operations is distinguishable from multiple FU's solely by the limitation of admitting one trigger per cycle.

Late-incoming operands. For some operations a non-triggering operand is allowed to arrive later than the triggering operand. The pipeline model captures this behaviour.

7.3.Constraints

1. If there are Hardware Operations that can access memory in a Function Unit, they must all access the same Address Space. Bank access, if necessary, is resolved at run time.
2. The inputs and outputs of a Hardware Operation cannot share the same Port.

8. Register Files

A register file (RF) is an array of registers that can be read or written through the same set of ports. Within an RF, each register has a unique name (index).

8.1.Parameters

Register File parameters	
RF-1	RF type: non-reserved, reserved, volatile.
RF-2	Bit width of the RF.
RF-3	Number of register in the RF.
RF-4	List of RF-ports.
RF-5	Binding between sockets and RF-ports.
RF-6	Maximum number of concurrent writes. If multiple concurrent writes to the same register in the register file is done, the stored value is undefined.
RF-7	Maximum number of concurrent reads of registers in the register file.
RF-8	Latency of registers, when used as guard terms.

8.2.Remarks

***RF-1*:** Three types of RF types are distinguished:

1. Non-reserved RF's: the registers in these RF's are available for general use. **Only this RF type is supported in TCE v1.0.**
2. Reserved RF's: the registers in these RF's are typically used to let the processor communicate with external modules in a more orderly fashion. These RF's contain registers that are available with restrictions for general allocation. These registers are assigned by the programmer and have global scope. Dataflow optimisations and reordering of definition and uses can be applied with some special restrictions.
3. Volatile RF's: the registers in these RF's are marked as volatile and can typically be used for register-mapped I/O. These RF's contain registers that are not available for general allocation. These registers are assigned by the programmer and have global scope. No dataflow optimisation or reordering may be applied to definitions and uses of volatile registers.

***RF-2*:** The bit width of the registers is specified per RF.

RF-3*:** Register files can have any number of ports. The number of ports is limited by ***RF-6 *and* ***RF-7***.

***RF-4*:** For each port the connected socket is independently specified.

***RF-8*:** Number of cycles (in addition to the global latency, described in Section 11) it takes before a register of the RF can be used in a guard expression. This is either 0 or

1, and by default 0. That is, guard latency is restricted only by the global guard latency (the control unit pipeline stage the guard is read at).

8.3.Constraints

1. Each RF input (output) port is connected to only one input (output) socket.
2. Each bidirectional port is connected to two sockets, one input and one output
3. Two or more ports in the same RF cannot be connected to the same socket. Each port of the RF is connected to a different socket. The port, however, can share the socket with ports *of other units* (Function Units, Register Files, etc.).
4. RF ports can be accessed by multiple buses at the same time, thus the same port can be read by multiple moves reading the same register at the same cycle. In case the register index in the multiple moves differs, the result is undefined.

9. Immediate Support

TTA Immediate support makes it possible to store constant numbers into the instructions of a TTA program. Currently, the specifications in this section assume bus-programmed target TTA processors.

9.1.Parameters

Two types of immediates are supported:

1. **Inline immediates.** An inline immediate is encoded in the instruction field normally used to encode the source address of a transport. The field belongs to the same move slot where the transport that reads the immediate is encoded. For this reason, an inline immediate is related to the bus that transports it. Support for inline immediates is defined by the following parameters:

Inline immediate parameters	
II-1	List of buses that can transport inline immediates.
II-2	Bit width of the longest inline immediate that can be encoded in each move slot.
II-3	Extension mode (sign or zero).

2. **Long immediates and Immediate Units.** Long immediates are encoded in one or more fields of the instruction stream. Immediate units are optional components that store constants from the instruction stream and make them accessible to the processor datapath just like any other programmable data source. An immediate unit is a register file whose register contents are determined by the long immediates encoded in the instruction stream, thus written only by the control unit directly. A single instruction can contain multiple long immediates, each destined to a different Immediate Unit. Thus, Immediate Unit is a Register File with only read ports exposed to the data path, and a single write port connected directly to the control unit (implicit in implementation only, not visible in ADF).

   The parameters that describe an immediate unit are:

Immediate Unit parameters	
LI-1	Number of immediate registers.
LI-2	Bit width of the Immediate Unit.
LI-3	Extension mode (sign or zero).
LI-4	Binding between sockets and Immediate Unit ports.
LI-5	Latency.
LI-6	*Instruction templates* that encode long immediates.
LI-7	Minimum bit width of the instruction fields that encode (part of) a long immediate.

9.2.Remarks

Inline Immediates:

***II-1*:** For each bus inline immediate support is specified independenly. It is not required that every bus support inline immediates.

***II-3*:** The immediate value is extended to the bit width of the data bus in which it is encoded.

Long Immediates:

***LI-1*:** Immediate Units are register files dedicated for holding immediate values that can be written only by instruction templates.

***LI-2*:** The bit width of the Immediate Unit registers is equal for all registers of a unit.

***LI-3*:** If the number of bits of a long immediate encoded in the instruction fields is smaller than the bit width of the Immediate Unit register, extension takes place. The extension mode must be specified for every Immediate Unit.

***LI-5*:** On certain implementations, long immediates may have to be encoded in instructions that precede the instruction containing the earliest move that reads the long immediate. For potentially higher performance, it is possible to encode long immediates in the same instruction that contains the earliest move that reads the long immediate. This possibility (latency zero) is termed *immediate bypassing*. This parameter is not supported in TCE v1.0. The latency is always fixed to 1. Thus, the next instruction after the instruction that defines the long immediate is the earliest one that can read the new value.

***LI-6*:** An instruction template specifies how immediate bits are encoded in an instruction (which fields of it contain the bits) and the destination IU register. A template can write to several immediate registers, as long as each register belongs to a different IU. A template cannot write multiple registers of the same destination IU at the same time. All immediate registers that are not defined by a template retain their value. There can be several instruction templates.

Stateless long immediate support: Under certain conditions, long immediates are not part of the machine state, and do not have to be really stored in an Instruction Unit. The conditions for this to happen are: (1) each IU has one register only; (2) every instruction template writes every IU.

Empty instruction templates: The empty instruction template represents the encoding format of instructions that don't contain long immediate bits or any immediate register-write action. The empty template is declared explicitly, as a template declaration without instruction field declarations. If there are no IU's, then the empty template is the only possible template, and is implied.

***LI-7*:** The number of bits that define (part of) a long immediate is defined explicitly for each instruction field that is used by a template, and is independent from the number of bits that the same instruction field has to encode transports on the bus. This prevents a subtle dependency between the instruction encoding and the target architecture.

9.3.Constraints

1. At least one bus must support inline immediates.

2. Every Immediate Unit must specify at least one instruction template.
3. Each port is connected to one socket; the socket must be an output socket.

10. Address Space

An address space defines a range of legal addresses that can be accessed by memory-accessing operations. Each address space is totally independent from the other address spaces. An address space consists of one or more memory banks. Memory banks can be single-ported or multi-ported.

Typically, the memory system of an embedded processor has two or three address spaces, one for the instruction memory and one or more for data memory.

The architecture specification does not define how concurrent accesses to a memory address space are implemented (whether by multi-banking, by multi-porting or by a combination of both). The maximum number of concurrent accesses to a memory address space is implied by the number of Function Units that access that memory.[3]

10.1. Parameters

Address Space parameters	
AS-1	Bit width of the minimum addressable word.
AS-2	Legal range of addresses (minimum and maximum address).

10.2. Remarks

***AS-1*:** Undefined minimum addressable word is allowed.

Memory banks. The banking of the memory is transparent to the architecture and is outside the scope of this specification. However, knowing the number of memory banks and their mapping to the address space can be useful to balance the accesses across the memory banks and achieve performance gains.

10.3. Constraints

If the bit width of the minimum addressable word is undefined, then only the Global Control unit (see Section 11, GC-7) can refer to that Address Space.

[3] Although it does not belong to the architecture definition, information such as the number of memory banks that form a memory address space and the address interleaving scheme can be very useful to help balance the accesses across the memory banks and achieve performance gains.

11. Global Control

Unlike the other processor components, the Global Control Unit has only one instantiation.

11.1.Parameters

Global Control parameters	
GC-1	Supported control flow operations.
GC-2	Input and output ports (operands and results of operations).
GC-3	Special port for the return address.
GC-4	Bit width of input and output ports.
GC-5	Binding between sockets and ports.
GC-6	Latency of the transport pipeline.
GC-7	Memory address space accessed by the fetch unit.
GC-8	Latency of guard expression term evaluation.

***GC-1*:** The processor may support jumps and call to subroutines with three different addressing modes: page-relative, PC-relative, absolute. A special kind of operation, *trap*, is used to raise software exceptions.

***GC-3*:** The RA register can be read and written though the RA port.

***GC-4*:** For each port the bit width is specified independently.

***GC-5*:** For each port the connected socket is independently specified. Several ports (even belonging to different units) can share the same socket. Thus, sockets are not necessarily tied to a single port.

***GC-6*:** The latency of the transport pipeline affects the latency of all operations that change the program flow.

***GC-7*:** The memory bank(s) that contains the program. May be a dedicated instruction memory or may be shared with data.

***GC-8*:** The number of cycles it takes to evaluate a guard expression from the given source (GPR or FU output port). This latency is independent on the number of cycles it takes to bring the output from a given unit (RF or FU) to the guard evaluation logic block.

12. Future Enhancements

The following requirements are identified, but have not addressed in this version of the ADF specification.

12.1.Hierarchical TTA

A single “master” TTA controls several “sub-TTAs” that are treated like coprocessors. The master may have to just provide the starting address of the data to be loaded by the sub-TTA, and initiate execution. The sub-TTA will then start running independently. The program in a sub-TTA might be pre-loaded.

What is the architectural support required for a hierarchy of TTAs?

Part 2: XML Specification

This part provides a complete normative description of the ADF format in XML.

13. Format Conventions

13.1.Aliases

Aliases are not allowed in the ADF format.

The motivation for not allowing aliases is that (1) aliases are *ad hoc* in nature and represent exceptions to default syntax, making the format more difficult to remember and more complex to parse; (2) an alias introduces an alternative way to specify the same piece of information contained in a standard declaration; (3) the exact information conveyed by an alias depends on the other ADF declarations.

13.2.References

All ADF elements that can be referenced by other elements are referenced by name and must be uniquely and explicitly identified by a “name” attribute. Implicit identification of elements (based on their declaration order, for example) is not allowed.

The main motivation for not allowing implicit identifiers is that it would complicate maintenance. For example, a change in the declaration order of an element list would require updating all references to the elements whose position in the list has changed.

13.3.Empty elements

Elements with default contents or empty contents are not optional. Allowing optional elements in these cases is rejected because it weakens the validation of ADF contents.

However, since the ADF format supports incomplete architecture definitions, certain types of elements can be omitted (mostly, elements for which multiple instances are permitted). Thus, missing elements, when permitted, simply mean missing data and incomplete definition, never a default value.

13.4.Naming convention for XML tags

The following rules and guidelines are applied for defining tag names of XML elements and attributes.

1. All tag names are in lower case.

2. In names made of multiple words, single words are separated by a hyphen.

3. Local tag names do not refer to the enclosing scope and should be kept to a minimum size.

4. A local tag name is informative and unambiguous within its local scope, but not necessarily unique across the entire ADF.

14. Validation

The XML description of an ADF undergoes three levels of verification.

1. Basic syntactical verification (*well-formedness*, in XML parlance).
2. *Validity* verification of the XML document within the XML processor.
3. *Validity* verification of the XML document that depends on external data bases, necessarily within client applications.
4. *Validity* verification of the object model, within the client applications.

Well-formedness is common to all XML documents and is not discussed here. For information, the reader is referred to: W3C, “Extensible Markup Language (XML) 1.0 (Second Edition)”, http://www.w3.org/TR/2000/REC-xml-20001006.

The *validity* verification of the XML document is restricted to those constraints that must be valid *even on an incomplete ADF.* Thus, an ADF may be a valid XML document, and yet represent a target processor that cannot be synthesised or cannot run any program.

A special group of validity constraints cannot be included in ADF because they require additional information that is stored in external files. Such constraints are termed “external constraints” and are listed below the XML constraints.

The *validity* verification of the processor object model takes place in the client application and takes into account high-level constraints on the template architecture that cannot be validated until the ADF is complete. In other words, the verification of the object model comprises all those constraints that is convenient to relax while the ADF specification of a target processor architecture is incomplete, to allow saving and reloading ADF’s of half-designed processor configurations by a client application.

Generic examples of this type of constraints are:

- Minimum number of elements in a variable-sized list of elements.
- Complex reference constraints (like unique chaining of bus segments).

The *validity* constraints that apply to the object model may be implemented in the object model itself or in the clients of the object model. These constraints are out of the scope of this document; their precise extent and nature depends on the client applications and is not uniquely defined. Certain clients may enforce stricter constraints simply because they do not support some of the “degrees of freedom” allowed in the templated architecture described by ADF.

There are several levels of “validity strictness” in the verification of the object model. For example, it does not make sense to allow a target architecture configuration with unconnected resources (e.g., a bus with no socket connections), but a valid target processor for such architecture could be synthesised. A processor architecture without a fundamental operation such as ‘add’ or ‘jump’, however, is not a valid architecture for a target processor.

15. General Constraints

The following constraints and definitions apply to elements or attributes of more than one declaration block.

15.1. Unique Name Attribute

Certain elements are identified by a mandatory *name* attribute termed Unique Name Attribute. The value of such *name* attribute can be used as a reference to the element from other elements within the same enclosing scope.

The Unique Name Attribute has the following properties.

1. The value of the attribute is a string that matches the following regular expression: [a-zA-Z][0-9a-zA-Z_:]*

2. Unless otherwise stated, the value of the attribute of an element must be unique for all elements of the same type within the scope of the enclosing element.

16. ADF Declaration

An ADF file is an XML file. Its root element is called “adf”.

The ADF file is combined, either by external reference or using an inline block, with an optional description of the binary encoding map that must be used to generate machine code (instruction bit vectors) for the target architecture.

```
<adf . . . > . . . </adf>
```

The *adf* element can contain only the following elements: *bus*, *socket*, *function-unit*, *register-file*, *immediate-unit*, *bridge*, *address-space*, and *global-control-unit*. All these top-level elements are optionals and can appear multiple times, except *global-control-unit*, which can appear at most once.

The top-level elements are described in the following sections.

Version Information

Version information is stored in the form of attributes of the top-level declaration element:

```
<adf version=”1.1” required-version=”1.0”>
. . .
</adf>
```

The *version* attribute is mandatory, while the *required-version* attribute is optional. If it is not specified, then its default value equals *version* value.

17. Top-Level Declaration

The little-endian option declaration has the following format:

```
<little-endian/>
```

This element is optional, and its existence determines the endianess of the architecture. ADFs with this element are little-endian, while ADFs without this element are big-endian.

18. Transport Bus Declaration

A transport bus declaration has the following format:

```
<bus name="string">
  <width> number </width>
  <guard> . . . </guard>
  . . .
  <segment name="name">
      <writes-to> segment </writes-to>
  </segment>
  . . .
  <short-immediate>
      <extension> string </extension>
      <width> number </width>
  </short-immediate>
</bus>
```

The *bus* Unique Name Attribute identifies the bus in the rest of the ADF. Names of immediate slots and names of transport bus slots share the same name space.

Note that the sockets connected to the bus are not specified in this declaration. Connections are specified in the socket declaration (Section 20).

A transport bus declaration contains four types of elements:

1. The element *width* gives the maximum bit width of the data transported.

2. A *guard* element specifies a guard expression supported on the transport bus for predicated data transports. A bus declaration may contain multiple guard elements. The *guard* element is described in Section 19.

3. A *segment* element specifies a bus segment. A bus declaration may contain multiple segment elements. The *segment* Unique Name Attribute identifies the segment within its bus. Combined with the Unique Name Attribute of its bus, it identifies the segment in the rest of the ADF. The *segment* element contains one mandatory element *writes-to*, which gives the segment that is driven by this segment.

4. The element *short-immediate* defines how inline immediates are supported by the bus and contains two mandatory elements: *extension* and *width*.
    a. The *extension* element gives the extension mode applied to the inline immediate word when it is less wide than the bus that transports it. The extension is an enumeration type consisting of two strings: “sign” and “zero”.
    b. The element *width* gives the number of bits of inline immediates that are encoded in the source field of the instruction slot.

XML validity constraints:

1. A *bus* element can contain only the following elements: *width*, *guard*, *segment*, *short-immediate*.

2. Elements *width* and *short-immediate* must appear once and once only.

3. Element *segment* is optional and may appear multiple times.

4. The element *width* must be a positive integer number.

5. The value of the *name* attribute of *segment* element must be unique among all segment declarations of a transport bus.

6. One and only one *segment:writes-to* element must be empty.

7. If not empty, the only valid contents of a *segment:writes-to* element are a string that represents the name of a segment declared in the same bus.

8. A segment can be referred to in no more than one *writes-to* element of another *segment* element.

9. Both elements *extension* and *width* must appear once and once only inside the *short-immediate* element.

10. The element *short-immediate:width* must be an integer number between zero and the width of the bus (inclusive).

11. The only valid contents of the *short-immediate:extension* element are the strings “sign” and “zero”. Any other string is not valid.

12. At most one *guard* element can contain element *unconditional* (see Section 19).

19. Guard Support Declaration

Each guard support declaration is enclosed in a *guard* element:

```
<guard>
    . . .
</guard>
```

and specifies a unique guard expression for a transport bus.

Explicit unconditional execution is represented with two special guard expression elements. The guard support declaration in this case contains either of the following elements:

```
<always-true/>
<always-false/>
```

The expression in the guard support declaration consists of one or two terms. A one-term guard expression can be of two types: *simple-expr* or *inverted-expr*. Also a two-term expression can be of two types: *and-expr* and *or-expr*. This results in the following four elements inside the *guard* declaration block:

```
<simple-expr> . . . </simple-expr>
<inverted-expr> . . . </inverted-expr>
<and-expr> . . . </and-expr>
<or-expr> . . . </or-expr>
```

The terms for conditional execution can be of two types: *bool* and *unit*. The *bool* element contains a reference to a register that belongs to a register file. The contents of the referenced register are used to compute the conditional value of the guard expression. A register reference consists of two parts: the register file name and the index of the register within the register file:

```
<bool>
    <name> regfile </name>
    <index> index </index>
</bool >
```

The *unit* element contains a reference to the output port of an FU. The value read from the referenced port is used to compute the conditional value of the guard expression. A port reference consists of two parts: the FU name and port name:

```
<unit>
    <name> unit </name>
    <port> port </port>
</unit>
```

A two-term guard expression (*and-expr* or *or-exp*) combines two one-term guard expressions with a logical operator. Each one-term guard expression can be either a *simple-expr* or an *inverted-expr*. For example:

```
<guard>
  <and-expr>
    <simple-expr>
      <bool>
```

```
        <name> rf_bool </name>
        <index> 0 </index>
      </bool>
    </simple-expr>
    <inverted-expr>
      <unit>
        <name> fu_cmp </name>
        <port> res </port>
      </unit>
    </inverted-expr>
  </and-expr>
</guard>
```

XML validity constraints:

1. A *guard* element must contain one and only one element in the following set: *always-true*, *always-false*, *simple-expr*, *inverted-expr*, *and-expr*, *or-expr*.

2. Elements *simple-expr* and *inverted-expr* must contain one term.

3. Elements *and-expr* and *or-expr* must contain two terms. These terms can be any combination of *simple-expr* and *inverted-expr*.

4. The conditional execution elements *simple-expr* and *inverted-expr* must contain exactly one element in the following set: *bool*, *unit.*

5. A *bool* element must contain two elements: *name* and *index*.

    a. The *name* element must be the valid name of a declared RF.

    b. The *index* element must be a nonnegative integer number, and must be a valid register index in the RF, that is, smaller than the RF size.

6. A *unit* element must contain two elements: *name* and *port*.

    a. The *name* element must be the valid name of a declared FU.

    b. The *port* element must represent the valid name of a port declared within the Function Unit referenced by *name* element.

20. Socket Declaration

A socket declaration has one of the following formats:

```
<socket name="string">
  <reads-from> . . . </reads-from>
  . . .
</socket>
```

```
<socket name="string">
  <writes-to> . . . </writes-to>
  . . .
</socket>
```

The *socket* Unique Name Attribute identifies the socket in the rest of the ADF.

A socket declaration contains either *reads-from* or *writes-to* elements. An element *reads-from* specifies the bus (segment) that this socket reads from. An element *writes-to* specifies the bus (segment) that this socket writes to.

Both *reads-from* and *writes-to* elements have the following format:

```
<bus> bus </bus>
<segment> segment </segment>
```

The *bus* element contains a reference to the bus read or written by this socket. The *segment* element contains a reference to the bus segment read or written by this socket.

Definition. A socket is a *binding socket* with respect to two bus chains if it refers to one bus on each chain.

XML validity constraints:

1. A *socket* element can contain only elements *writes-to* or *reads-from*.
2. Elements *writes-to* and *reads-from* are optional and mutually exclusive.
3. Elements *writes-to* and *reads-from* may appear multiple times.
4. Elements *writes-to* and *reads-from* must contain one and only one element *bus*, and one and only one element *segment*.
5. The only valid contents of elements *writes-to:bus* and *reads-from:bus* are the strings that represent valid names of declared buses.
6. The only valid contents of elements of *writes-to:segment* and *reads-from:segment* are the strings that represent valid names of segments of the bus specified by the corresponding *bus* element.
7. A combination of bus and segment names may not appear in more than one *writes-to* or *reads-from* element of a socket declaration.

21. Bridge Declaration

A bridge declaration has the following format:

```
<bridge name="string">
  <reads-from> bus </reads-from>
  <writes-to> bus </writes-to>
</bridge>
```

The *bridge* Unique Name Attribute identifies the bridge in the rest of the ADF.

A bridge declaration contains two elements:

1. The element *read-from* contains a reference to a bus and specifies that the bridge reads from the given bus.

2. The element *writes-to* contains a reference to a bus and specifies that the bridge writes onto the given bus.

XML validity constraints:

1. A *bridge* element contains only elements *writes-to* and *reads-from*.

2. Elements *reads-from* and *writes-to* must appear once and once only.

3. The only valid contents of the *writes-to* and *reads-from* elements are the strings that represent valid names of declared buses.

4. The same bus name may not appear in the *writes-to* and in the *reads-from* element.

5. At most two *bridge* elements may refer to the same bus in their *writes-to* elements.

6. At most two *bridge* elements may refer to the same bus in their *reads-from* elements.

7. Taking all *bridge* elements together, the busses referred to in their *writes-to* and *reads-from* elements must form a unique, acyclic chain of busses.

22. Function Unit Declaration

A Function Unit declaration has the following format:

```
<function-unit name="string">
    <port name="string">
        <connects-to> socket </connects-to>
        . . .
        <width> number </width>
        <triggers/>
        <sets-opcode/>
    </port>
    . . .
    <operation> . . . </operation>
    . . .
    <address-space> name </address-space>
</function-unit>
```

The *function-unit* Unique Name Attribute identifies the Function Unit being declared in the rest of the ADF.

A Function Unit declaration contains four element types:

1. The element *port* declares a port used to interface the Function Unit with the rest of the processor. The *port* Unique Name Attribute identifies the port within the FU. A port declaration consists of the following elements:
    a. One or two *connects-to* elements. Each element contains a reference to a socket, and specifies that the FU port reads from (or writes to) one or more transport buses through the given socket. Input ports are connected to a socket that reads from a bus. Output ports are connected to a socket that writes to a bus. Bidirectional ports are connected to two sockets, one writing onto and one reading from a bus.
    b. The element *width* defines the bit width of the port. The bit width must be a positive integer number.
    c. An optional, valueless element *triggers*. If present, it specifies that reading (or writing) this port starts the execution of a new operation.
    d. An optional, valueless element *sets-opcode*. If present, it specifies that reading (or writing) this port selects the operation to be executed.
2. The *operation* element declares the implementation of an operation of the architecture's instruction set. The *operation* element is described in Section 23.
3. The *address-space* element declares the address space that can be accessed by the FU. The *address-space* element is described in Section 26.

XML validity constraints:

1. A *function-unit* element can contain only the following types of element: *port*, *operation*, *address-space*.

2. Element *address-space* must appear once and once only.

3. Elements *port* and *operation* are optional and may appear multiple times.

4. The only valid contents of *port:connects-to* elements are strings that represent valid names of declared sockets.

5. The element *port:width* is mandatory, and must appear only once.

6. The value of the *port:width* element must be a positive integer number.

7. The element *port:connects-to* is optional, and may appear once or twice within each *port* element. If it appears twice, the sockets referred to must not have the same direction.

8. The element *port:sets-opcode* must appear in at most one port.

9. A port element that contains the *sets-opcode* element must also contain the *triggers* element.

10. The only valid contents of a nonempty *address-space* element are strings that represent valid names of declared address spaces. An empty *address-space* indicates that the Function Unit does not access a memory bank.

External validity constraints:

1. The *width* element of the address-space declaration (see Section 26) referred to by the *address-space* element must not be empty.

23. Hardware Operation Declaration

A Hardware Operation declaration block is local to a Function Unit declaration and has the following format.

```
<operation>
  <name> operation </name>
  <bind name="number"> port </bind>
    . . .
  <pipeline> . . . </pipeline>
</operation>
```

A Hardware Operation declaration contains the following elements:

1. The mandatory *name* element identifies the operation being declared with a string of characters.
2. A variable number of *bind* elements. Each element binds one of the inputs or outputs of the base operation with one of the ports of the Function Unit that contains the Hardware Operation. A *bind* element contains a mandatory *name* attribute. The value of the *name* attribute is a number, and identifies one of the inputs/outputs of the base operation. The value of the *name* attribute must be unique among all *bind* elements within the *operation* element scope.
3. The element *pipeline* declares the execution pipeline of the operation. The *pipeline* element is described in Section 24.

XML validity constraints:

1. An *operation* element can contain only the following types of element: *name*, *bind*, *pipeline*.
2. Elements *name* and *pipeline* must appear once and once only.
3. The value of the *name* element must be unique among all *operation* elements of a Function Unit declaration.
4. The only valid contents of the *bind* elements are the strings that represent valid names of ports declared in the containing *function-unit* element.
5. It is not allowed for two *bind* elements to specify the same port.
6. The *bind:name* attribute must be a positive numer and must be unique among all the *bind* elements of an *operation* declaration.

External validity constraints:

1. The value of the *name* element must be a string of characters that matches one of the operation names allowed for the target architecture.
2. The value of the *name* element is a string that matches the following regular expression: [a-z_][0-9a-z_]*
3. There must be one *bind* element for each of the operation inputs.

24. Execution Pipeline Declaration

An execution pipeline declaration block is local to a Hardware Operation declaration and has the following format:

```
<pipeline>
    <resource name="string">
        <start-cycle> number </start-cycle>
        <cycles> number </cycles>
    </resource>
    . . .
    <reads name="number">
        <start-cycle> number </start-cycle>
        <cycles> number </cycles>
    </reads>
    . . .
    <writes name="number">
        <start-cycle> number </start-cycle>
        <cycles> number </cycles>
    </writes>
    . . .
</pipeline>
```

An execution pipeline declaration contains *resource*, *reads*, and *writes* elements. All elements are optional.

A resource is a part of a pipeline that is used when executing the operation. The pipeline declaration shows the occupation for each resource cycle by cycle. Resources are identified by the value (a character string) of the mandatory attribute *name*. The scope of a resource is the Function Unit declaration. Thus, two *resource* elements having the same *name* value refer to the same processor resource, whether the element appears in the same *pipeline* element or in different *pipeline* elements of operation declarations in the same Function Unit. Resource usage elements do not have to be declared before being referenced. The order of the resource elements is not important.

Elements *reads* (*writes*) indicate the inputs (outputs) of the operation that are accessed and thus, indirectly though the *bind* declaration of section 23, which ports of the FU are read or written. By specifying *writes* elements in different cycles, it is possible to define independent latencies between the triggering operand and each operation output. Elements *reads* and *writes* are identified by a value (a number) of the mandatory attribute *name*. This number identifies one of the operation inputs (outputs), therefore the scope of these elements is the *operation* declaration.

All three types of elements contain the following elements:

1. The mandatory *start-cycle* element gives the first cycle in which this resource is used relative to the beginning of the operation. The start cycle must be a nonnegative integer number. Cycle ‘0’ is the cycle in which the operation is triggered, that is, the operand bound to the triggering port is written.
2. The mandatory *cycles* element indicates the number of cycles this resource is occupied.

The following example shows the pipeline of a floating-point divide operation with a 2-cycle unpipelined unpack, a 2-cycle pipelined pack phase and a 10-cycle

unpipelined computation phase, a cycle which is overlapping with the unpack phase.

```
<pipeline>
    <resource name=”s1”>
        <start-cycle> 1 </start-cycle>
        <cycles> 2 </cycles>
    </resource>

    <reads name=”1”>
        <start-cycle> 1 </start-cycle>
        <cycles> 2 </cycles>
    </reads>

    <reads name=”2”>
        <start-cycle> 1 </start-cycle>
        <cycles> 2 </cycles>
    </reads>

    <resource name=”s2”>
        <start-cycle> 2 </start-cycle>
        <cycles> 10 </cycles>
    </resource>

    <resource name=”s3”>
        <start-cycle> 12 </start-cycle>
        <cycles> 1 </cycles>
    </resource>

    <resource name=”s4”>
        <start-cycle> 13 </start-cycle>
        <cycles> 1 </cycles>
    </resource>

    <writes>
        <start-cycle> 13 </start-cycle>
        <cycles> 1 </cycles>
    </writes>
</pipeline>
```

Some operations can be completed by the end of the same cycle in which the operation is triggered. In such zero-latency operations, the results are available in the next cycle, and accesses to inputs and outputs are concentrated in cycle zero of the pipeline declaration.

The following example shows the pipeline of a zero-latency operation with two input operands and one output (for example, a bitwise OR).

```
<pipeline>
    <resource name=”s1”>
        <start-cycle> 0 </start-cycle>
        <cycles> 1 </cycles>
    </resource>

    <reads name=”1”>
        <start-cycle> 0 </start-cycle>
        <cycles> 1 </cycles>
    </reads>

    <reads name=”2”>
        <start-cycle> 0 </start-cycle>
        <cycles> 1 </cycles>
```

```
    </reads>
    <writes name="3">
        <start-cycle> 0 </start-cycle>
        <cycles> 1 </cycles>
    </writes>
</pipeline>
```

XML validity constraints

1. The only valid value for the name attribute of *reads* and *writes* elements is a string that represents a positive integer.
2. Element *start-cycle* must contain a nonnegative integer.
3. At least one of the *resource* or *reads* elements must have the *start-cycle* element containing “1” or “0”.
4. Element *cycles* a positive integer.
5. **Nonoverlapping uses of the same resource.** Two *resource* elements may not have the same name attribute if *start-cycle* element of the later usage contains a number that is smaller than the sum of values in *start-cycle* and *cycles* elements of the other *resource* element.
6. **Canonicity**: Two *resource* elements may not have the same name attribute if *start-cycle* element of the later usage contains a number that is equal to the sum of values in *start-cycle* and *cycles* elements of the other *resource* element.

External validity constraints:

1. There must be at least one *reads* element for each of the operation inputs.
2. There must be at least one *writes* element for each operation output bound to a port.

25. Register File Declaration

A register file declaration has the following format.

```
<register-file name = "string">
    <type> . . . </type>
    <size> number </size>
    <width> number </width>
    <max-reads> number </max-reads>
    <max-writes> number </max-writes>
    <guard-latency> number </guard-latency>
    <port name="string">
        <connects-to> socket </connects-to>
        . . .
    </port>
    . . .
</register-file>
```

The *register-file* Unique Name Attribute identifies the register file being declared in the rest of the ADF.

A register file declaration contains the following elements:

1. The *type* element indicates how the RF is used. A RF can be used for general register allocation (normal), for custom, user-controlled register allocation (reserved) or for user-controlled I/O communication (volatile). Only 'normal' type is supported by TCE v1.0.
2. The *size* element gives the number of registers contained in the register file and must be a positive integer number.
3. The *width* element gives the bit width of the registers and must be a positive integer number.
4. The element *max-reads* gives the maximum number or ports that can read a registers all in the same cycle.
5. The elements *max-writes* gives the maximum number of ports that can write registers all in the same cycle.
6. The *guard-latency* element declares the local latency of guard terms evaluated out of the registers of this register file.
7. The element *port* declares a port used to interface the register file with the rest of the processor. The *port* Unique Name Attribute identifies the port within its register file. A *port* element contains up to two *connects-to* elements.
   A *connects-to* element contains a reference to a socket and specifies that the register file port reads from (or writes to) one or more transport buses through the given socket. A port connected to a socket that reads from a bus is an input port. A port connected to a socket that writes to a bus is an output port. A port connected to two sockets, one writing onto and one reading from a bus, is a bidirectional port.

XML validity constraints:

1. A *register-file* element can contain only elements of the following types: *type*,

size, *width*, *max-reads*, *max-writes*, *port*.

2. Elements *type*, *size*, *width*, *max-reads*, max-writes, *max-rw* must appear once and once only.
3. Element *port* is optional and may appear multiple times.
4. The only valid contents of the *type* element are the strings “normal”, “reserved”, and “volatile”.
5. Elements *size* and *width* must contain a positive integer number.
6. The *max-reads* element must contain a nonnegative integer number.
7. The *max-writes* element must contain a nonnegative integer number.
8. A nonempty *guard-latency* element must contain an integer 0 or 1. An empty *guard-latency* element indicates that the latency of the guard terms from this Register File is zero, or that there are no guard terms that read a register of this Register File.
9. The only valid contents of *port:connects-to* elements are strings that represent valid names of declared sockets.
10. Each *port:connects-to* element must be unique within the entire RF.
11. The element *port:connects-to* is optional; it may appear once or twice within each *port* element. If it appears twice, the sockets referred to must not have the same direction.
12. The count of architectural ports connected to the register file must not exceed the *max-reads* and *max-writes* properties of the register file. That is, there can be a maximum of *max-reads* read ports and *max-writes* write ports in the register file. The *max-reads* and *max-writes* properties are to store the architectural information in “deattached register files” in certain databases in which ports do not provide direction information.

26. Address Space Declaration

An address space declaration has the following format.

```
<address-space name="string">
    <width> number </width>
    <min-address> number </min-address>
    <max-address> number </max-address>
</address-space>
```

The *address-space* Unique Name Attribute identifies the address space being declared in the rest of the ADF.

The address space declaration contains the following mandatory elements:

1. The *width* element specifies the bit width of the minimum addressable word (equal to the memory bank bit width). An empty element specifies an undefined bit width.
2. The *min-address* element specifies the lowest address in this address space.
3. The *max-address* element specifies the highest address in this address space.

XML validity constraints

1. An *address-space* element can contain only elements of the following types: *width*, *min-address*, *max-address*.
2. All elements must appear once and once only.
3. If not empty, the *width* element must contain a positive integer number.
4. Both *min-address* and *max-address* elements must contain a nonnegative integer number.
5. The value of *min-address* must be lower than the value of *max-address*.

27. Global Control Unit

The Global Control Unit declaration has the following format.

```
<global-control-unit name="string">
    <port name="string">
        <connects-to> socket </connects-to>
        <width> number </width>
        <triggers/>
        <sets-opcode/>
    </port>
    . . .
    <special-port name="string">
        <connects-to> socket </connects-to>
        . . .
        <width> number </width>
    </special-port>
    . . .
    <return-address> port </return-address>
    <ctrl-operation>
        <name> string </name>
        <bind name="number"> port </bind>
        . . .
        <pipeline> . . . </pipeline>
    </ctrl-operation>
    . . .
    <address-space> name </address-space>
    <delay-slots> number </delay-slots>
    <guard-latency> number </guard-latency>
</global-control-unit>
```

The *global-control-unit* Unique Name Attribute identifies the Global Control Unit in the rest of the ADF and must be unique in the Function Unit name space.

The Global Control Unit declaration contains elements of the following types:

1. A *port* element declares a port used to interface the Global Control Unit with the rest of the processor. The element *port* has exactly the same format of element *port* described in Section 22.
2. A *special-port* element declares a port used to interface a special-purpose register of the Global Control Unit with the rest of the processor. The *special-port* Unique Name Attribute identifies the port within the FU. A *special-port* element consists of the following elements:
    a. One or two *connects-to* elements. Each element contains a reference to a socket, and specifies that the FU port reads from (or writes to) one or more transport buses through the given socket. Input ports are connected to a socket that reads from a bus. Output ports are connected to a socket that writes to a bus. Bidirectional ports are connected to two sockets, one writing onto and one reading from a bus.
    b. The element *width* defines the bit width of the port. The bit width must be a positive integer number.
3. The *return-address* element declares which special port is connected to the

return address register.

4. The *ctrl-operation* element specifies which control transfer operations (e.g. jump, call, trap) are supported. These operations are identified with a unique name. The specification of *operation* element described in Sections 23, 24 apply, unchanged, to *ctrl-operation* element.
5. The mandatory *address-space* element declares the address space where the program is stored.
6. The mandatory *delay-slots* element declares the number of instructions executed following a conrol flow instruction (e.g., JUMP). The count depends on control unit implementation details such as count of stages of the transport pipeline.
7. The mandatory *guard-latency* element declares the global latency of guard expression evaluation.

XML validity constraints

1. Element *global-control-unit* can appear at most once.
2. The *global-control-unit* element can contain only elements of the following types: *port*, *special-port*, *return-address*, *ctrl-operation*, *address-space*, *delay-slots*, *guard-latency*.
3. Elements *address-space*, *return-address*, *delay-slots* and *guard-latency* must appear once and once only.
4. Elements *port*, *special-port* and *ctrl-operation* are optional and may appear multiple times.
5. The attribute *name* of a *port* and *special-port* element must be a unique string among all *port* and *special-port* elements in the Global Control Unit declaration.
6. The only valid contents of *connects-to* elements within elements *port* and *special-port* are strings that represent valid names of declared sockets.
7. The *width* element of *port* and *special-port* elements is mandatory, and must appear only once.
8. The *width* element of *port* and *special-port* elements must contain a positive integer number.
9. The element *connects-to* in *port* and *special-port* elements is optional, and may appear once or twice within each element. If it appears twice, the sockets referred to must not have the same direction.
10. The element *port:sets-opcode* must appear in at most one port.
11. If a port element contains the *sets-opcode* element, it must contain the *triggers* element, too.
12. The only valid contents of nonempty *return-address* element is a string that represents a valid name of a *special-port* contained in the Global Control Unit declaration.
13. The only valid contents of nonempty *address-space* element is a string that represents a valid name of a declared address space.
14. The value of the *delay-slots* element must be an integer number greater or equal to

zero.

15. The value of the *guard-latency* element must be a nonnegative integer number. If there is a guard term that reads a register of a Register File which has local guard latency of zero, the guard-latency element must be a positive integer number.

28. Immediate Unit Declaration

Immediate Unit Declaration has the following format.

```
<immediate-unit name=”string”>
   <extension> extension </extension>
   . . .
   <template name = ”string”>
       <slot>
         <name> slotname </name>
         <width> number </width>
       </slot>
       . . .
   </template>
   . . . (any register file element) . . .
</immediate-unit>
```

The *immediate-unit* Unique Name Attribute identifies the immediate unit being declared in the rest of the ADF. An immediate unit is architecturally a register file. Thus, all properties supported by a Register File Declaration, except defining write ports are supported also by Immediate Unit Declarations.

Additionally, long immediate declaration contains the following elements:

1. The *extension* element gives the extension mode applied to the long immediate when it is encoded in a number of bits smaller than the bit width of the destination immediate register and written to a register in the immediate unit. The extension is an enumeration type consisting of two strings: “sign” and “zero”.
2. The element *template* declares (part of) an instruction template that writes to an IU register. A *template* element is identified by its mandatory name throughout the ADF. If an instruction template writes into several IU’s, the template definition is “distributed” across the destination IU’s. Parts of the same instruction template are specified in separate *template* elements with the same name, one in each destination IU. Any register of a destination IU can be written: the register choice orthogonal with respect to the template and the destination IU. A *template* element may contain *slot* elements.
    a. A *slot* member specifies an instruction field in which (part of) a long immediate is stored. The bits of the long immediate defined by a template declaration are concatenated from different slots in the same order in which slots are declared. A *slot* element contains one *name* element and one *width* element.
        i. The *name* element identifies the instruction field (transport bus slot, described in Section 18 or immediate dedicated slot, described in Section 29) in which (part of) the long immediate is stored.
        ii. The *width* element gives the minimum bit width of the instruction field as well as the number of significant bits of the long immediate that are encoded in the instruction field.

XML validity constraints

1. In addition to any element supported by a *register-file* declaration, an *immediate-unit* element can contain only elements of the following types: *extension*, *template*.
2. The only valid contents of *extension* element are the strings “sign” and “zero”. Any other string is not valid.
3. The value of the *name* attribute of *template* element must be unique among all template declarations of an immediate unit.
4. The only valid contents of *slot:name* elements are strings that represent valid names of declared buses or valid names of declared immediate slots.
5. A slot name may not appear in more than one *slot:name* element within a *template* element.
6. The property *max-writes* derived from *register-file* declaration is always and by default 1 for Immediate Unit Declarations. This is because IU is a register file that is written by the control unit, thus needs one write port architecturally, when considering IU as a component separate from the rest of the architecture. However, that write port is not visible to the data path. Therefore, the validity check of count of architectural write ports should be less than equal to the *max-writes* does not hold for immediate units.

29. Immediate Slot Declaration

An immediate slot declaration has the following format:

```
<immediate-slot name="string"/>
```

The *immediate-slot* Unique Name Attribute identifies the dedicated immediate slot in the rest of the ADF. Names of immediate slots and names of transport bus slots share the same name space.

Appendix

Abbreviations

ADF	Architecture Definition File
ADFF	Architecture Definition File Format
ASP	Application Specific Processors
COTS	Commercial Off-The-Shelf
FU	Function Unit
GPR	General-Purpose Register
IU	Immediate Unit
MDF	Machine Definition File
MDFF	Machine Definition File Format
PC	Program Counter
RF	Register File
TNO	Netherlands Organisation for Applied Scientific Research
TNO-FEL	TNO Physics and Electronics Laboratory
TTA	Transport Triggered Architecture
TUNI	Tampere University
XML	eXtensible Markup Language

Glossary

Bridge	Storage element that allows to pipeline a data transport between two transport busses.
Bus segment	Smallest independently programmable unit of a transport bus that can carry out a data transport.
Function Unit	Processor block that can carry out operations of the TTA.
Immediate	Link-time constant bit pattern stored in the instruction stream.
Immediate Unit	Processor block that stores immediates.
Instruction template	Encoding format of a TTA instruction in a given processor implementation.
Move	Atomic unit of a TTA instruction that specifies a data transport. Also, the data transport itself.
Register File	Processor block containing a set of registers for temporary storage of data.
Transport bus	Programmable bus of a TTA processor that can carry out data transports.
Unit	One of: Register file, Function Unit, Immediate Unit or Global Control Unit.

Dependencies Between ADF Elements

The following diagram shows the cross-references existing between the top-level elements of ADF. The tail of the arrow is attached to the (sub)element that contains the reference, the head is attached to the referenced (sub)element. Thicker links without arrow depict containment, not references. For example, Guard elements are contained by Bus elements. Internal dependencies such as those between operation declarations and FUPort elements are not shown.