TTA Program Exchange Format

Program File Format for a New TTA Design Framework

Authors: *A. Cilio (Tampere University of Technology)*

TAMPERE UNIVERSITY OF TECHNOLOGY

1. Summary

This document contains the complete reference specification of the TTA Program Exchange Format (TPEF) for a new TTA design framework. TPEF allows to store in a persisting form a TTA Program in any stage of its processing, from unscheduled code to TTA code scheduled for a specific TTA target processor.

Document History

Version	*Date*	*Author*	*Comment*
0.1		A. Cilio	First draft of the document.
0.2	13/07/03	A. Cilio	Second draft, corrections from M.Lepistö and P.Jääskeläinen proofreading
0.3	17/10/03	A. Cilio	Modifications to immediate and empty instruction specifications (TEXT section).
0.4	02/12/03	A. Cilio	Added documentation of instruction element annotations.
0.5	12/02/04	A. Cilio	Added program profile section. Section identifier is now 16-bit long. Meaning of sh_size for (u)data sections changed.
0.6	27/02/04	A. Cilio	No overload of sh_aspace field; sh_link extended to 16 bits. Changed references to sections in section LINENO.
0.7	26/04/04	A. Cilio	Revision after meeting at NRC, Helsinki. Added symbol table section. Completed relocation section. Renamed TEXT section to CODE.
0.8	05/05/04	A. Cilio	Revision after M.Lepistö comments. Rejected chunk number. Added relocation diagram. Added table for sh_info member.
0.9	10/05/04	A. Cilio	Added general properties and limits. Rejected support PC-relative relocation. Clarified page-relative relocations.
0.9.1	30/07/04	A. Cilio	Revised introduction and relocation section. Corrected error in relocation example.
0.10	05/08/04	A. Cilio	Changed slightly the meaning of asp_wsize member. Added sh_info data (instruction word size) for code section. Added document section about sequential architecture.
0.11	09/10/04	A. Cilio	Changed specification of sh_info for code section.
0.12	09/01/05	A. Cilio	Complete specification of move guards. Changed FU input port and instruction element annotations.
0.13	05/02/05	A. Cilio	Redefined format of immediates. Redefined source and destination format for GPR's, variables and special registers.
0.14	10/02/05	A. Cilio	Corrections to move format. Added program entry symbol.
0.15	21/02/05	A. Cilio	Corrected guard and move type specifications. Added name to address spaces.
0.16	19/05/05	A. Cilio	Introduced MVS_BRIDGE type for move sources. Minor updates to take bridges into account.
0.17	04/08/05	M. Lepistö	Added STT_PROCEDURE symbol type.
0.18	12/10/05	A. Cilio	Added MRT_PORT. Corrections to format of MVT_UNIT.
0.18.1	30/11/05	A. Cilio	Disabled instruction size and encoding in TUT_TTA.

Table of Contents

2. Introduction

TPEF is the name of the binary file format used to define TTA programs for a given target processor architecture. Such architecture must conform to a transport-triggered architecture template. TPEF supports multiple TTA templates.

Part of the TPEF format specification, such as code sections and relocation information, depends on the TTA template. This document describes also the template-specific part of the specification that applies to the TTA Codesign Environment architecture template.

A TPEF file describes a TTA program at different levels of abstractions: from object code containing unresolved references to unscheduled linked code to (possibly partially) scheduled TTA code for a specific TTA target processor. For a complete definition of the TTA architecture template, see the specification document of the Machine Definition Format [1]. For a description of the architecture template conventions, see the Architecture Specification document [2].

Part 1: Requirements

This part of the document specifies the requirements of TPEF.

3. Requirements

Representation of sequential and parallel TTA code. The TPEF instruction format can represent sequential and parallel code. This requirement can be achieved adopting a "minimal" description of the scheduled code and an idealised architecture for the sequential code that is consistent with concrete target architectures.

Size efficiency. Variable encoding is a must to achieve size-efficient representation of TTA code, given the discrepancy in amount of data necessary to describe empty or "filled" TTA instructions, sequential and parallel moves.

Full support for parallel TTA code. The information contained in a TPEF file is sufficient to unambiguously reconstruct, when combined with the appropriate target processor architecture description of a MDF file, the precise parallel code.

Support for code relocation. Relocation requires that all references to instructions or data, that is all constant addresses (either immediate source fields of moves or initialisation data of variables) can be identified. A relocation table is useful for this purpose, but might be insufficient. To simplify reference lookups, immediates should be stored in the TPEF file in a consistent way, independent of how they are actually encoded in the instruction stream expected by the target processor. Link and reference resolution functionality is strictly limited to pure sequential code

Independence from external target architecture description. The information in a TPEF file completely defines a parallel TTA program; the Machine Definition File that describes the target architecture is optional. TPEF specification of processor resource usage may be incomplete. The MDF can be used to complement the program data with details otherwise left unspecified (if any).

Parallel and sequential TTA code can be mixed in the same file. This requires that the hardware resource categories whose usage is represented in TPEF files must have resource identifiers for sequential (unscheduled) code. These resource identifiers are reserved to sequential code, and cannot be used to represent real hardware resources of a target processor. This approach makes it possible to represent partially scheduled instructions, where, for example, part of the resources required the carry out the data transport are assigned (and specified in the TPEF file) while other resources are left unspecified (that means, the identifiers reserved to unscheduled code are used).

Support for multiple program sections of a type. This requirement implies unique section identifiers for each type of section.

Support for linkage and reference solving. This implies that not all references in a TPEF file contain data, and that unresolved, external symbols must be supported. Linkage implies also possible auxiliary data structure to organise library files.

Support for debugging information. Symbolic information is stored as symbol table entries in STABS format. Either *ad-hoc* symbol table sections or the same symbol table used for other program symbolic information can be used. Some source-level debugging information may be difficult or even impossible to maintain in scheduled code, because operations and moves are scattered, and original source lines are lost.

Part 2: Binary Format Specification

This part provides the normative specification of the TPEF binary representation. The TPEF format is extensible. New sections can be added, provided that the basic section header information conforms to the TPEF format specifications.

4. Format Overview

The information stored in a TPEF file is organised in section. Simply put, a section is a collection of information items of similar type. A TPEF file consists of a file header and a variable number of sections. Each section is accompanied by a section header, which provides identification and file-accessing information. Some section headers may not have a corresponding raw-data section. The following table shows the file's organisation.

file header
section header 1
...
section header *n*
section 1
...
section *m*

Scope of the TTA Program File Format. Format specification is organised in two levels: the common part and the architecture-specific part. The common part of this specification describes the format of the data structures common to every TPEF file and the base format of certain file sections. This part dictates the minimum requirements that an implementation must satisfy in order to read/write well-formed TPEF files. This part of the specification also describes the correct way to extend the information contained in common data structures without losing backward compatibility. Other file sections, notably the sections containing program instructions and data, and the sections containing debugging information, are specific to the target architecture and the toolset that generates the file. These sections are not strictly part of the format specification, but represent extensions thereof. The format described for these sections is specific for a target architecture, the new TTA Design Framework developed at the Tampere University of Technology, and can be used as a guideline for defining TPEF specification of other architecture templates.

Byte order. The byte order of any number stored in TPEF is big-endian, independent of the host computing system that created the TPEF file.

5. File Header

The file header provides a information to:

1. Unambiguously identify the file as a TPEF file
2. Locate and correctly interpret the contents of the file

Field	*Bytes*	*Description*
fh_id	10	File format identification mark.
fh_arch	1	Architecture (template) of the TTA code stored.
fh_type	1	The type of TTA program contained in this file.
fh_shoff	4	File offset to the first section header.
fh_size	2	Size (in bytes) of this file header (including *fh_id*).
fh_shsize	2	Size (in bytes) of a section header. All entries have same size.
fh_shnum	2	Number of section headers.
fh_shstrtab	4	File offset to section header of the string table that contains section name strings.

The file format identifier *fh_id* contains the following bytes:

Byte	*Value*
id_magic[0]	0x7F
id_magic[1]	0x54 'T'
id_magic[2]	0x54 'T'
id_magic[3]	0x41 'A'
id_magic[4]	0x2D '-'
id_magic[5]	0x50 'P'
id_magic[6]	0x46 'F'
id_magic[7]	0x00
id_version	Version number of this file format
id_size	Size of this identifier structure

The current version number contained in *id_version* structure is 1.

The file offset past the last byte of the *fh_id* structure should always be obtained by reading *id_size*. In this way, it will be possible to increase the size of the structure

without unnecessarily impairing compatibility. See Resolved Issues, item 3.

TPEF file types. The *fh_type* member indicates the type of TTA code stored in this file according to the following table:

fh_type	*Value*	*File Type*
FT_NULL	0x00	Illegal/Undefined file type.
FT_OBJSEQ	0x04	Sequential TTA object code.
FT_PURESEQ	0x05	Sequential TTA code without unresolved symbols.
FT_LIBSEQ	0x06	Collection of sequential TTA object files (library).
FT_MIXED	0x09	Partly scheduled code.
FT_PARALLEL	0x0D	Fully scheduled or mixed fully scheduled/sequential code.

All values not shown in table are reserved and cannot be used.

Sequential TTA code (FT_OBJSEQ), in the context of TPEF files, is defined as *completely unscheduled* TTA code, possibly containing unresolved references to procedures or variables. Sequential TTA libraries (FT_LIBSEQ) contain the type of code of sequential TTA object files. Libraries may also contain additional information to speed up and assist the linkage process. Pure sequential TTA code (FT_PURESEQ) is completely unscheduled TTA code that [discuss] contains no unresolved references and needs no linkage. Mixed TTA code (FT_MIXED) contains partially scheduled instructions, that is, instructions where only some transports have all resources assigned. Parallel TTA code (FT_PARALLEL), in the context of TPEF files, is fully scheduled TTA code, where each instruction has all the needed machine resources assigned. A TPEF file of TTA parallel code may contain pure sequential TTA code in some procedures.

What changes with different file types is the amount of information stored, not the format in which this information is stored. The file type indicates restriction on the expected contents of a TPEF file. It must be considered just a “hint” for TPEF file readers.

Support for different architecture templates. It is foreseen that the TTA Program Exchange Format will be used by different TTA design toolkits, possibly based on a different basic architecture template. The *fh_arch* field defines the basic TTA (possibly templated*) architecture* of the file. The purpose of this field is to make it easier to identify and ignore TPEF files for unknown TTA architecture templates. These files should probably appear as partly corrupted TPEF file for the architecture considered, hence the necessity for a clear marker.

The value of *fh_arch* determines how the program sections (instruction and data sections) are encoded. Existing types of auxiliary sections should maintain the same format independent of the architecture (template). The values currently defined for *fh_arch* are listed in the table below.

Name	*Value*	*Description*
NOARCH	0x00	Illegal/Undefined architecture
TTA_MOVE	0x01	Move design framework – Delft University of Technology
TTA_TUT	0x02	TTA template for the new TTA Codesign Environment – Tampere University of Technology
TDS_TI	0x03	TTA Design Studio architecture by S. Pekarich – Texas Instruments

The specification of the program section format contained in this document is applicable to the new TTA template developed at TUT.

fh_shstrtab contains a file offset to a special *string table section* (see below), which contains the names of the sections. This string table is not required. If not present, then *fh_shstrtab* should contain zero.

6. Section Header

Sections hold the bulk of file information: instructions, data, symbol table, relocation information, and so on.

The section header describes the internal organisation of a file section, and allows a TPEF reader to locate and interpret the raw data it contains. All section headers have the same size.

Every section in a TPEF file must have a section header. The converse is not true (see description of *sh_flags* structure).

Field	***Bytes***	***Description***
sh_name	4	Section offset to the name of the section (zero: no name available).
sh_type	1	Section type.
sh_flags	1	Section flags.
sh_addr	4	If a program section (see below), starting memory address of the section, zero otherwise.
sh_offset	4	File offset to the section data area.
sh_size	4	Size (in bytes) of the section data area stored in this file.
sh_id	2	Section identification number.
sh_aspace	1	Section address space identifier (zero if not applicable).
–	1	Padding byte. Must contain zero.
sh_link	2	Section identifier link (see below).
sh_info	4	Section-specific information.
sh_entsize	4	Size (in bytes) of a section entry.

The name of the section, if defined, is specified as a string table index in *sh_name.* The file offset to the string table is specified by file header's member *fh_shstrtab.*

The sh_type field uniquely defines the *purpose* of the section according to the following table:

Section	***Value***	***Description***
SHT_NULL	0x00	Inactive section.
SHT_STRTAB	0x01	Section holds a string table.

Section	*Value*	*Description*
SHT_SYMTAB	0x02	Section holds symbol entries.
SHT_DEBUG	0x03	Section holds information for symbolic debugging.
SHT_RELOC	0x04	Section holds relocation entries.
SHT_LINENO	0x05	Section holds source code line numbers.
SHT_NOTE	0x06	Section holds information that marks file in some way.
SHT_ADDRSP	0x07	Section contains address space information.
SHT_MDF	0x08	Section contains the complete machine description.
SHT_LIBTAB	0x09	Section contains auxiliary information for library files.
SHT_MR	0x0A	Section holds machine resource entries.
SHT_PROF	0x0B	Section holds program profile data.
SHT_CODE	0x81	Section holds program instructions.
SHT_DATA	0x82	Section holds program's data initialisation values.
SHT_UDATA	0x83	Section describes the program's uninitialised data area.

Values 0xA to 0xF and 0x84 to 0x8F are reserved for future extensions.

For a description of the section types listed in the table, see below.

The sections in a TPEF file are divided in two broad categories:

1. *Program sections.*
2. *Auxiliary sections*.

The most significant bit (0x80) of *sh_type* tells whether the section is a program (the bit is '1') or an auxiliary section (the bit is '0'), and its use is reserved.

A program section represents the contents of (part of) a TTA program. For a TTA program to be run, the information contained in program sections has to be loaded in the memory system of the target processor. Hence, program sections need additional information about the addressing space where they should be loaded. Address space information is stored in an address space table. The address space table is a section that must be present in any TPEF file that contains program sections.

Program sections do not contain the exact memory image of the program, nor do they dictate how the program is actually encoded in the memory. See Section 22 for more information on data encoding.

There are two types of program sections: sections containing program instructions (code sections) and sections containing program data (data sections).[1] More than one

[1] More types of program sections may be added in the future.

section of both types can be present.

Auxiliary sections contain auxiliary information referring to one or several program sections. For example, a *Line Number* Section refers to one code section (see below), while a String Table or a global Symbol Table may contain data referring to several program sections.

The *sh_flags* member contains a number of flags that define additional properties of the file section that apply to any section type:

Flag	***Value***	***Description***
sf_nobits	0x80	Section is not initialised/not stored in this file (1).
sf_vlen	0x40	Section contains entries with variable length.
–	0x2F	Reserved bit mask for future flags. Must be set to zero.

The flag *sf_nobits* specifies if the data area of the section is stored in the file. A section's data may be not stored when the section header is sufficient to convey all (available) information about the section of when the information is simply not available.

Member *sh_addr* gives the start address of the memory area corresponding to the image of a program section. The number stored in *sh_addr* is the absolute address of the first MAU of the memory image.

Member *sh_size* contains the size of the section in bytes. Unless the section has the *sh_flags* flag *sf_nobits*=1, the section occupies *sh_size* bytes in the file. A section for which *sf_nobits*=1 may have a non-zero size, but it occupies no space in the file.

Each section has a section identifier number, defined in member *sh_id*. This number is unique across all file sections contained in the file, irrespective of their type. The number zero is reserved for a special section of type SHT_NULL, described in section.

If the section is a program section, *sh_aspace* contains a number that identifies its address space. If the section is an auxiliary section, *sh_aspace* is not used and must contain zero.

The section link member *sh_link* contains the section identification number of another file section. The type of the referenced section depends on the section type and is listed in the following table.

Section	***Type of section pointed at by sh_link***
SHT_NULL	SHT_NULL
SHT_STRTAB	SHT_NULL
SHT_SYMTAB	SHT_STRTAB

Section	*Type of section pointed at by sh_link*
SHT_DEBUG	SHT_SYMTAB
SHT_RELOC	SHT_SYMTAB
SHT_LINENO	SHT_SYMTAB
SHT_NOTE	?
SHT_ADDRSP	SHT_STRTAB
SHT_ADF	SHT_MR
SHT_LIBTAB	?
SHT_MR	SHT_STRTAB
SHT_PROF	SHT_CODE
SHT_CODE	SHT_MR
SHT_DATA	SHT_NULL
SHT_UDATA	SHT_NULL

Member *sh_info* holds information that applies to the whole section and whose interpretation depends on the section type. For example, auxiliary sections such as Relocation and Line Number sections are tied to one program section (in the latter case, a CODE program section). In these sections, the *sh_info* member contains a reference to the program section.

The table below describes the use of *sh_info* member (if any) for each section type. For a more detailed description the reader is referred to detailed section specifications. When *sh_info* is not used, its value must be zero.

Section	*Data stored in sh_info member*
SHT_NULL	Unused.
SHT_STRTAB	Unused.
SHT_SYMTAB	Index of the first entry with nonlocal link scope [DISCUSS]
SHT_DEBUG	Program section to which debugging information applies.
SHT_RELOC	Program section that contains the references to relocate.
SHT_LINENO	Code section where instructions of the source lines is stored.
SHT_NOTE	Unused.
SHT_ADDRSP	Unused.
SHT_ADF	Unused.
SHT_LIBTAB	?
SHT_MR	Unused.

Section	*Data stored in sh_info member*
SHT_PROF	Unused?
SHT_CODE	Size (in MAU's) and/or type of encoding of instruction word.
SHT_DATA	Unused.
SHT_UDATA	Unused.

Shared Auxiliary Sections. [discuss: this part is messy] String table, debug, and symbol table sections may be tied to a single program section or not. When they are not tied to one particular section, these sections are assumed to be shared by multiple program sections, and the *sh_link* member must contain SHT_NULL. In this case, all program sections must share the auxiliary section.

File sections may contain pointers to strings (encoded as file offsets to string table positions). TPEF allows the flexibility of separate or combined string tables for each file section.

Some sections hold a table of fixed-size entries, such as a symbol table. For such a section, the *sh_entsize* member gives the size in bytes of each entry. The member contains 0 if the section does not hold a table of fixed-size entries. In that case, the *sf_vlen* flag of *sh_flags* member must be set to 1.

Each memory address space is characterised by properties such as the "natural" and the minimum addressable word size and the alignment. This information is defined in a special section called *address space table* (see Section 10 of this document).

7. TPEF sections

This section describes properties that are shared by several (possibly all) TPEF sections.

7.1. Section Format and Constraints

A TPEF file is organised into sections. A section is described by two items:

- A section header with fixed format, described in document Section 6.
- An optional data area or payload, whose format depends on the section type.

Section data areas have the following properties:

- A data area occupies contiguous sequences of bytes in the file.
- Data areas of different section do not overlap.
- It is possible for bytes of the file to be outside any section data area. Such bytes may be located between data areas, between section headers and the first data area or past the last data area. Such data is called *inactive space* and its contents are out of the scope of the TPEF.

7.2. Undefined References to Section Elements

Certain sections, if their payload is stored in the file (*sf_nobits*=0) cannot have an empty data area. These sections must always have an element reserved for special purposes, the *null element*.

The null element is needed for those section elements that can be referred to from other sections, and for which an undefined reference is possible.

Currently, null elements apply to the following sections:

- Address space table: reserved entry (identified by zero) for nonapplicable references to address space.
- String section: null byte '\0' representing undefined strings.
- Symbol table: zero-index entry reserved for undefined symbol references.

8. General Properties and Constraints

This section describes properties and constraints that apply to the TPEF file in its entirety.

8.1. Range Limits

TEPF files contain several instances of integer numerical data types that represent references (addresses, identification codes, indices or offsets). Depending on the reference type, a different bit width and thus a different range of values is admitted. These range limits are an essential part of the format, they pose hard constraints on the capabilities of TPEF.

The following table lists the upper range limit of file-wide reference types that are encoded as integer numbers.

Reference	*Range Limit*	*Description*
–	0xFFFFFFFF	Section offsets (section-specific references).
–	0xFFFFFFFF	Data sizes (in bytes, sh_size and section-specific).
sh_offset	0xFFFFFFFF	File offset.
sh_addr	0xFFFFFFFF	Address in an address space.
r_symbol	0x00FFFFFF	Symbol table entry index.
sh_id	0x0000FFFF	Section identification code.
sh_aspace	0x000000FF	Address space identification code.

8.2. Bit Width of Encoded References

No matter what the bit width of the encoded reference and thus the range limit is, the following rule applies to all references encoded in a TPEF file: Any reference of a given type must be encoded in a number of bits sufficient to represent any reference of that type. In other words, an *n*-bit reference data type must always be encoded in *n* (or more) bits.

8.3. Undefined References

If a reference to a given type of data items may be undefined or not applicable, a special instance of the data type must be reserved for such references.

For example, references to sections may be undefined, not applicable or irrelevant. Therefore, TPEF defines a special section type and a reserved identification number for the Null Section. Other examples of such reserved instances, where the data item is a section element, are described in section 7.2.

9. Null Section

The Null Section represents an invalid section and is used for section references in cases where no section is applicable. A section header for the Null Section must be always present and has the value zero assigned its *sh_id*. The following table shows the values of section header of the Null Section.

Field	***Value***	***Notes***
sh_name	–	Any name string is allowed, including the null string.
sh_type	0	SHT_NULL
sh_flags	0x80	Section data area is not stored in this file.
sh_addr	0	Not applicable.
sh_offset	0	Not applicable.
sh_size	0	Size is always zero.
sh_id	0	Reserved section identification number.
sh_aspace	0	Not applicable.
sh_link	0	No link = link to Null Section itself.
sh_info	0	Ignored, must be set to zero.
sh_entsize	0	Not applicable.

10. Address Space Table section

In a TTA program, different program sections can be stored in independent memories. As a result, a TTA program may contain identical addresses that refer to entities stored in different memories. Each memory defines an independent address space. Since an address may refer to completely different entities in different address spaces, relocation information must take address spaces into account. The Address Space Table is stored in a special section (*sh_type* SHT_ADDRSP) and holds the main properties of the address spaces used by the program sections of the file. There must be one and one only Address Space Table section in a TPEF file. The Address Space Table contains one entry for each address space.

Field	***Bytes***	***Description***
asp_id	1	Unique identification number of this address space.
asp_mau	1	Minimum addressable unit bit width of this memory.
asp_align	1	Alignment (in *asp_mau* units) of the natural word of this memory.
asp_wsize	1	Size (in *asp_mau* units) of the *natural* word of this memory.
asp_name	4	Section offset to string table entry for the name of this element.

The *asp_id* value zero has a special meaning. The first address space entry is assigned identification number zero. This entry is described later.

The *asp_mau* field can contain zero. In this case, the bit width of the words is not known. If *asp_mau* is zero, the size of the natural word given by *asp_wsize* field must be 1. In practice, only the MAU of an address space that contains program instructions can have undefined bit width.

The addresses stored in a program section and in its relocation section are expressed in *asp_mau* units. All sizes contained in symbol table are also assumed to be in *asp_mau* units.

The *asp_align* field is necessary to verify that the memory addresses to natural word (see below) are not misaligned.[2]

Example. Alignment of a natural word. A memory with 8-bit MAU (*asp_mau* =8) could hold 4-MAU natural data words (*asp_wsize* =4) with 2-MAU alignment constraint (*asp_align* =2). The address 0x7F02 can adddress a natural word, whereas 0x7F01 cannot.

The "natural" word is the data word of the bit width for which the target architecture gives full (or most extended) support. In practice, the natural width of the data section is the width closest to the bit width of general-purpose registers. In address spaces that contain only instructions, the natural word is the minimum addressable unit. The alignment constraints of instruction addresses (when applicable) are determined by

[2] TPEF clients may choose to ignore misaligned words. Ultimately, it is responsibility of the client that manipulates a TTA program to verify that all addresses of the target memory system are valid.

the size of the word (member *sh_info* of code section header) and are unrelated to *asp_align* and the concept of natural data word.

Null Address Space. The first address space entry has *asp_id* = 0 and does not define a real address space. It is a placeholder. Its identification number is used in all cases in which a field should contain a reference to an address space, but no actual address space is applicable in the context. For example, no address space is applicable to auxiliary sections. This address space entry is always present in the address space table and has the following values:

Field	***Value***	***Notes***
asp_id	0	Reserved identifier: no real address space.
asp_mau	0	Minimum addressable word size not applicable.
asp_align	0	Alignment of the natural word not applicable.
asp_wsize	0	Size of the *natural* word not applicable.
asp_name	0	Null address space has no name: null string.

All program sections must have a non-zero address space identifier.

11. Code section

Code sections (*sh_type* SHT_CODE) contain program code, that is, TTA instructions. A TTA instruction specifies one or more data transports that take place in the same clock cycle. Each data transport is programmed by means of a simple operation called *move*.

Instruction size in memory image. The format of instructions stored in a TPEF code section has nothing to do with the binary encoding in the target memory system. Consequently, the size of the instructions stored in a code section does not correspond to the actual length in the memory image of the program. However, the instruction length is necessary in order to correctly compute the target instruction corresponding to an instruction address as stored in a program section. For this reason, the instruction length (or a reference to the algorithm used to compute it) is stored in field *sh_info* of code section header according to the following table.

sh_info field	***Bytes***	***Description***
in_size	0–1	Length (in MAU's) of instructions in program image; zero if length is undefined or variable.
in_enc	2	Instruction encoding algorithm; 0x00 means undefined.
–	3	Padding byte. Must be zero.

Member *in_size* contains the length of the TTA instructions, expressed in MAU's. If the instruction encoding algorithm produces instructions of variable length then *in_size* contains zero.

If *fh_arch* is TUT_TTA, *in_size* is ignored, and treated as if it were '1'.

Member *in_enc* contains a number that identifies the algorithm used to encode the TTA instructions in the target memory. Values 0x00 to 0x7F are reserved to fixed-length encoding algorithms; values 0x80 to 0xFF are reserved to variable-length encoding algorithms. Value 0x00 is reserved to undefined encoding. If the encoding algorithm is undefined, then the instruction size must be defined.[3]

In *fh_arch* is TUT_TTA, *in_enc* must be set to 0x00.

Instruction elements. A code section contains variable-length *instruction elements*. Each element represents either a move or an immediate, which is a constant value encoded in the instruction stream and used by a move. Several adjacent elements form a TTA instruction. Each element consists of two parts: a fixed-length *attribute member* and a variable-length, type-dependent part.

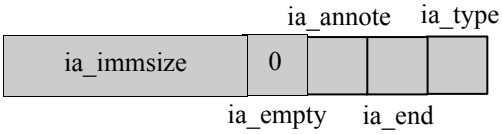
The attribute member *i_attr* discriminates immediate elements from move elements and specifies certain attributes. The attribute member is one byte long and is subdivided in a number of fields. The meaning of the attribute fields and their bit mask is described in the following table.

[3] If both *in_size* and *in_enc* are zero, then the instruction length is unknown and the code section cannot be fully interpreted.

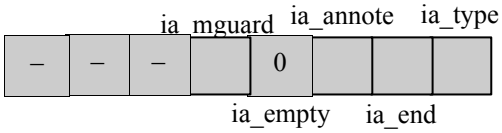
i_attr field	*Mask*	*Description*
ia_type	0x01	Type of element: immediate (1) or move (0)
ia_end	0x02	End-of-instruction (1)
ia_annote	0x04	Contains annotation (1)
ia_empty	0x08	Empty instruction (1)
ia_immsize	0xF0	Number of bytes used to encode the immediate
ia_mguard	0x10	Conditional move (1) or unconditional move (0)

Member *ia_immsize* gives the number of bytes (1–16) that are used in the TPEF file to encode the immediate value. This number does not necessarily reflect the exact bit width of the immediate, which doesn't have to be a multiple of 8. For example, the 20-bit immediate with binary representation 00000000 000011111111 (0xFF) could be stored in 1 byte instead of 3.

For immediate elements the format of the attribute member is the following:



For move elements the format of the attribute member is the following:



Unused fields are reserved for future use, and must be set to zero.

In-line annotations. Annotations are a mechanism to augment an instruction element with additional, arbitrary and application-dependent information without disrupting the normal working of other TPEF applications. All annotations that are not recognised by an application must be preserved unchanged. The *ia_annote* flag specifies whether the instruction element contains one (or more) variable-size, in-line annotations. An annotation consists of a number of bytes in the range 4–131 and has the following format:

Field	*Bytes*	*Description*
an_size	1	Length (in MAU's) of the payload of the annotation.
an_id	3	Identification code.
–	4..127	Free-format, variable-length payload.

Annotations consist of two parts: a fixed part and a free part. The contents of the free part represents the "payload" of the annotation and are out of the scope of the TPEF specification, and depend on the specific application that creates, reads or modifies those annotations. The common part consists of 2 fields. The first, *an_size*, contains

the number of bytes that make up the annotation payload and the continuation bit. When the continuation bit (mask: 0x80) is set to 1, the annotation is followed by another annotation. Member *an_id* contains a unique identification number that allows applications to recognise the annotations they define or use from annotations used by other applications. There is no formal mechanism to prevent clashes in the identification space, but the availability of over 16 million identification numbers should minimise the risk of annotations with conflicting identifiers. The identification numbers in the range 0x000000 – 0x00FFFF are reserved for future expansions of the TPEF standard.

Empty instructions are specified by means of a single instruction element with *ia_empty* and *ia_end* flags set to1. The type of the element given by *ia_type* is ignored. To represent a well-formed empty instruction, also the *ia_end* flag of the previous element must be set to 1. No other member of an empty instruction is stored (except, when applicable, in-line annotations).

Guarded moves are specified by *ia_mguard*=1. When *ia_mguard*=0, the move is always executed. The move fields *mv_grfu* and *mv_gndx* (described later on) are present only when the move is guarded.

11.1. Immediate Element

The Immediate Element specifies an immediate value (a run-time constant) used by one or more moves. The following table shows the members that make up an immediate element.

Field	***Bytes***	***Description***
i_attr	1	Instruction element attribute member.
im_dunit	1	Destination unit for an immediate field; 0 for in-line.
im_index	1	Destination identifier or index of the immediate register.
im_image	1..16	Bit vector that represents the immediate value.

In addition to the fixed-size attribute member *i_attr*, an immediate element contains three fields: *im_dunit*, *im_index* and a vector of *ia_immsize* bytes. Each byte of this vector contains 8 bits of the immediate value.

Immediates are not necessarily encoded in the target-architecture format. The format is 2's complement for integer numbers and IEEE-754 for floating-point numbers.

Every immediate has a destination. Such destination may be a physical register (such is the case of immediates containing large constants, or *long* immediates) or an identifier of the move source field where the immediate is actually encoded. In the second case, the immediate element must be stored in the same instruction where the move using it is stored (in other words, no intervening element can have *ia_end* flag set to 1), and must precede the move. The destination is specified by members *im_dunit* and *im_index*. The first identifies the destination immediate unit; the second identifies the destination register within the immediate unit. In-line immediates are a special case signalled by *im_dunit*=0x00; in this case, member *im_index* is a unique number that identifies the immediate within the TTA instrucition that contains it.

Example: Encoding of an immediate that contains the unsigned integer number 200:

i_attr	im_dunit	im_index	im_image[0]
00010001	00000000	00000101	11001000

member *im_dunit* signals that no (architecturally visible) physical register is assigned as destination of the immediate; this means that either the immediate bits are encoded directly in the source field of moves (in fully scheduled moves) or no destination register (and no instruction bits) has been assigned to this immediate. In this example, *imd_index*=5 uniquely identifies the source field of the move in current instruction where the immediate bits are encoded. This is the move that reads the immediate. The same index 5 appears in the source field of the move.

Example: Encoding of an immediate that contains the 4-byte number 0xABADF00D:

i_attr	im_dunit	im_index	im_image[0]	im_image[1]	im_image[2]	im_image[3]
01000011	00000011	10000010	10101011	10101101	11110000	00001101

An immediate element for a 4-byte long number takes 7 bytes. In this example, *ia_end*=1, thus the immediate is the last element of the current instruction. Member *im_dunit* specifies that the immediate destination is a register in the immediate unit identified by a machine resource entry of type MR_IU with index 3. Because this is a real entry, it implies that the immediate bits are encoded in (fields of) an instruction, rather than in a move source field. Member *im_index* specifies that the immediate is written into a physical register with index 2. The immediate element can be part of an instruction several cycles before the first use of the immediate, and the immediate may be used by several moves.

11.2. Move Element

A move element describes a move and consists of a variable number of members, shown in the following table.

Field	***Bytes***	***Description***
i_attr	1	Instruction element attribute member.
mv_bus	1	Move bus identifier.
mv_type	1	Move source/destination type fields.
mv_sunit	1	Move source unit.
mv_sndx	2	Move source register index/identifier.
mv_dunit	1	Move destination unit.
mv_dndx	2	Move destination register index/identifier.
mv_grfu	1	Move guard function unit/register file.
mv_gndx	2	Move guard index/identifier.

A *scheduled* move is part of a parallel instruction, which specifies several concurrent data transports. A fully scheduled move is assigned the processor components (such as registers, encoding bits, the transport bus and the connections) it needs to carry out its data transport. At the other end of the spectrum, an *unscheduled* move specifies only the source and destination register of the data transport. See section 23 "Unscheduled Target TTA" for details on the unscheduled (or "sequential") target TTA implied by TPEF architecture template identified by *fh_arch* = TTA_TUT. Details on the conventions and restrictions that apply to programs for TTA_TUT target architecture template can be found in [3].

To distinguish sequential TTA code from parallel code even within the same program section, a hardware component identification number is reserved for the transport bus, the function unit, and a number of register files (see below for details).

The move bus specifier *mv_bus* is simply a number that uniquely identifies a bus of the target processor. This number can express up to 127 (0x01 to 0x7F) transport busses. The values 0x80 to 0xFF are reserved for future extensions. (Probably, additional information on the transport resources used by the move.)

Member *mv_type* is divided into three fields that specify the type of source, destination and guard fields of the move.

mv_type field	***Mask***	***Description***
–	0x01	Unused bit
mvt_src	0x0E	Move source type
mvt_dst	0x30	Move destination type
mvt_grd	0xC0	Move guard type

The source field of move elements can specify one of the following:

1. a general-purpose register,
2. one of the results of an operation (and thus, implicitly, an output port of the function unit assigned to the operation),
3. a *special/reserved register* (and thus, implicitly, a function unit port),
4. a *long immediate register*,
5. a number that identifies an immediate encoded in the move source field,
6. a *bridge register* that contains the value on an adjacent bus in previous cycle.

The types of destination field are:

1. a general-purpose register,
2. one of the operands of an operation (or, equivalently, a function unit input port),
3. a *special/reserved register* (and thus, implicitly, a function unit port).

The following table shows how the type of source fields is encoded.

Value	*Encoding*	*Description*
MVS_NULL	0x00	Undefined source, not used.
MVS_BRIDGE	0x02	Bridge to adjacent transport bus.
MVS_RF	0x04	Variable or general-purpose register.
MVS_IMM	0x08	Immediate register or in-line immediate.
MVS_UNIT	0x0C	Operation output (function unit output port).

The following table shows how the type of destination fields is encoded.

Value	*Encoding*	*Description*
MVD_NULL	0x00	Undefined destination, not used.
MVD_RF	0x10	Variable or general-purpose register.
–	0x20	Illegal destination.
MVD_UNIT	0x30	Operation input (function unit input port).

There are four types of guard fields, determined by two flags. The following table shows how the flags are encoded.

Field	*Mask*	*Description*
mvg_inv	0x40	Guard of the move inverted (1) or not inverted (0).
mvg_rf	0x80	Guard source is a GPR (1) or a FU output (0).

Bit 0x01 of member *mv_type* is reserved for future expansion and must be set to zero.

The type of move source and destination specified by *mv_type* determines how source and destination members are interpreted. The source and the destination of a move are described by means of two members each: *mv_sunit*, *mv_sndx* for the move source, *mv_dunit*, *mv_dndx* for the move destination.

Every function unit, register file, immediate unit and bridge register in the target processor is assigned a unique identification number. Members *mv_sunit*, *mv_dunit* contain a number that identifies an instance of one of the following types of resources:

1. for general-purpose registers of any type: register file,
2. for inputs or outputs of an operation, or for FU ports: function unit,
3. for special registers: function unit,
4. for bridge registers: transport bus that writes the bridge register.

In addition, *mv_sunit* can specify also the immediate unit of an immediate source.

In case the source (destination) of a move is a result (operand) of an operation, or a FU port, *mv_sunit* (*mv_dunit*) gives the function unit the operation is assigned to.

Reserved units. The values 0x80 to 0xFF of register files are reserved to variable pools. Variable pools are not real register files; they contain a finite, unbounded

number of variables of predefined bit width (data type). See Sections 12 and 23 for details on the reserved identifiers for the architecture of unscheduled code.

The value 0x00 of *mv_sunit*, *mv_dunit*, for any resource type, is reserved and has a special purpose, described in Section 12.

The second member of the move source and destination define:

1. for variables and general-purpose registers: index of the register or variable,
2. for function units: numbers identify the machine resources (entries in MR section) of one of the following types: operand or the result of operation (MRT_OP), special register (MRT_SR), or port (MRT_PORT).

In addition, member *mv_sndx* can also define one of the following:

1. for in-line immediates: identifier contained in member *im_index* of the immediate element that defines the immediate value (see above),
2. for long immediates: index of the immediate register.

If *mv_sunit* specifies a bridge source, then member *mv_sndx* is unused.

The input and output identifiers of different implementations of the same operation don't have to be unique. Operation inputs and outputs corresponding to ports in different units can be identified by the same numeric code, if they belong to different implementations of the same operation. Similarly, port resources in different units can share the same numeric code and entry in the resource table. The meaning of the unit and operation input/output identifiers is fully specified by means of (mandatory) auxiliary information. See Section 12 for details.

Member *mv_grfu* identifies the unit (register file or function unit) that contains the data from which the result of the guard expression is computed. This number can express up to 255 register files or function units, including those reserved for unscheduled code.

Member *mv_gndx* identifies the source of the data used to compute the result of the guard expression. Depending on the type of unit it can represent:

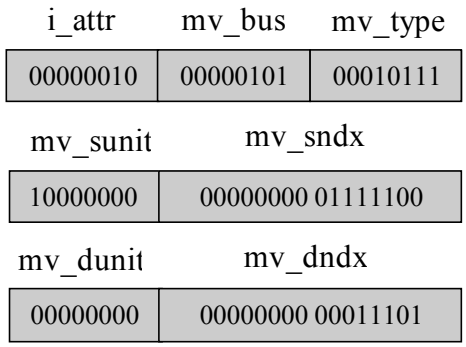
1. for register files: index of the register or variable,
2. for function units: a number (restricted to range 0x0000 – 0x7FFF) that identifies a result of an operation.

Examples. Move encoding.

Let us consider an instruction element containing the following unscheduled move:

```
r124 -> div.1
```

this move specifies a data transport from variable 'r124' to the first (index one) operand of the integer divide operation. Supposing that the operand index *mv_dndx* assigned to operation input 'div.1' is 29, the element takes up 9 bytes and is encoded as follows:


i_attr
mv_bus
mv_type
00000010
00000101
00010111
mv_sunit
mv_sndx
10000000
00000000 01111100
mv_dunit
mv_dndx
00000000
00000000 00011101

12. Processor Resource Table section

While TPEF does not require a complete description of the target architecture, parts of the program need to refer to processor components (machine resources, in instruction scheduling jargon). The Processor Resource Table is stored in a special section (*sh_type* SHT_MR) and contains *resource description elements*. Each element is identified by a number and describes a processor component or, indirectly, a part of it.

Machine resources are of different ypes. Each type has its own set of unique identification numbers. The following types of machine resources are recognized:

1. busses,
2. register files,
3. function units,
4. immediate units,
5. operation inputs/outputs,
6. special registers,
7. function unit ports.

A resource description element has the following structure.

Field	***Bytes***	***Description***
mr_index	2	Machine resource identification number.
mr_type	1	Machine resource type (see below).
mr_name	4	Element name, as section offset to string table entry.

The *mr_type* field specifies the type of machine resource and can have one of the following values.

Resource Type	***Value***	***Description***
MRT_NULL	0x00	Illegal undefined machine resource.
MRT_BUS	0x01	Transport bus.
MRT_UNIT	0x02	Function unit.
MRT_RF	0x03	Register file.
MRT_OP	0x04	Operation input or output.
MRT_IMM	0x05	Immediate unit.
MRT_SR	0x06	Special register.
MRT_PORT	0x07	Function unit port.

The string table where the names of the resources are stored is specified by the *sh_string* element of the section header.

Identifiers of type MRT_BUS are restricted to the range 0x0000 – 0x007F. The range of identifiers of types MRT_UNIT, MRT_RF and MRT_IMM is 0x0000 – 0x00FF.

The machine resource MRT_OP represents the combination of two pieces of information:

- an operation (opcode);
- the index of an operation input or output.

The input or output of an operation, given any target architecture compatible with the program, implicitly defines also the port of the function unit, through the binding declaration (see ADF specifications, [3]).

The machine resource MRT_SR represents special-purpose registers attached to a function unit. Special registers are identified by fixed, unique string names. The only special register recognised is the return address register (RAR), identified by the string 'return-address'.

The machine resource MRT_PORT represents a function unit port. Ports are identified by a fixed name string, which is unique within the function unit but not the target architecture. It is possible to "share" the same resource port entry to refer to several ports, as long as they belong to different function units.

Shared space of identification numbers. All resource types that are related to a function unit (operations, special registers, ports) share the same id space. It is not possible that, for example, an MRT_OP and a MRT_SR entry have the same *mr_index* value.s

Reserved identification numbers. The following resource identification numbers are reserved to the architecture of unscheduled TTA code (see Section 23 for details).

Resource Type	***Value***	***Description***
MRT_BUS	0x0000	Unassigned bus.
MRT_UNIT	0x0000	Universal function unit.
MRT_RF	0x0000	Illegal register file assignment.
MRT_RF	0x0080	Integer variable pool.
MRT_RF	0x0081	Boolean variable pool.
MRT_RF	0x0082	Floating-point variable pool.
MRT_IMM	0x0000	Signals an in-line immediate.

The register file identifier 0x00 is illegal.

References to reserved identification codes can be mixed (even in the same move element) with references to identification codes of normal resource description elements. Thus, TPEF code section can contain partially scheduled TTA instructions.

13. Initialised data section

A data section (*sh_type* SHT_DATA) contains the values with which program variables are initialised before the program is started.

The bits contained in a data section reflect the raw data encoding of a byte-addressed, big-endian architecture that represents integer numbers in 2's complement binary form, and real numbers in either 32 or 64-bit IEEE-754 standard floating-point representations. This data encoding may or may not match the data encoding of the target architecture. See Section 22 for more details on how program data should be accessed.

Member *sh_elsize* contains the size of the minimum addressable word of the section, expressed in bytes (see Section 22 for a description of how this value is computed).

The size of data sections is stored in *sh_size*, and is expressed in bytes. The size of the corresponding memory image in the target memory system, expessed in minimum addressable words, of *asp_mau* bits (see Address Space Table, Section 22), can be computed with the help of member *sh_elsize*.

For example: given a data section of an address space that is addresses by 12-bits words with a size (*sh_size*) of 824 bytes and with *sh_elsize* = 2, the number of MAU's stored in this data section is 412.

14. Uninitialised data section

A data section that contains uninitialised data[4] (*sh_type* SHT_UDATA) is typically not stored in TPEF files. Hence, *sh_flags* bit *sf_nobits* is expected to be set. In this case, the contents of *sh_offset* member are ignored.

Member *sh_elsize* contains the size of the minimum addressable word of the section, expressed in bytes.

The size of uninitialised data sections is stored in *sh_size*, and is expressed in bytes. To compute the corresponding number of minimum addressable words of *asp_mau* bits (see Address Space Table section for details), the size in bytes must be divided by the size of the section element, *sh_elsize.*

[4] In some file formats, e.g. BSD a.out and ELF, sections containing uninitialised data are termed "BSS" sections.

15. Symbol Table section

Symbols map strings to values (typically, names to addresses). The linker refers to the name of a symbol until its address has been assigned (resolved). Debuggers use symbol names to represent program information in a form that is more readable to humans. Symbols consist of a fixed-length entry in the symbol table and a variable-length name in a string table (see Section 18).
A symbol table entry holds information needed to locate program symbolic definitions and references. Symbol tables contain entries of various types. All are always referred to by their index into the symbol table section.
A symbol table entry has the following structure.

Field	***Bytes***	***Description***
st_name	4	Name of the symbol, section offset into the string table.
st_value	4	Value of the symbol.
st_size	4	Size of the symbol, in bytes. Zero if not defined or not applicable to the symbol type.
st_info	1	Type and binding attributes of the symbol.
st_other	1	Symbol value descriptor.
st_section	2	Section to which the symbol belongs.

The first entry of every symbol table section (index 0) is reserved to undefined symbol references. The symbol table entry for undefined references is described later.
The meaning of the value of a symbol table entry depends on the symbol type. For symbols that belong to code and data sections the value represents an address; for other symbols it may be arbitrary.
Member *st_size*, when nonzero, holds the size of the symbol, given as the number of bytes taken by the symbol in this file. The correspondence between bytes in the file and the actual size in the memory image is determined by the encoding rules described in Section 22. The size of certain symbols, such as instruction labels, is irrelevant and sometimes is not even unknown. For these symbols, *st_size* is always zero.
If a symbol with nonzero *st_size* belongs to a section whose data area is not stored in the TPEF file, then the size gives the number of bytes that the symbol would take if it were stored in the TPEF file.
Member *st_info* is subdivided in two: the upper half (4 bits) specifies the binding attributes of the symbol; the lower half specifies the symbol type.
The binding attribute determines the linkage scope of the symbol and how the linker must treat it:

Binding	***Value***	***Description***
STB_LOCAL	0x0	Symbol is only visible in the program sections that come from the file that contains its definition.

Binding	*Value*	*Description*
STB_GLOBAL	0x1	Symbol is visible to all program sections that have the same address space of the symbol.
STB_WEAK	0x2	Symbol visibility is equal to STB_GLOBAL symbols, but its linkage priority is lower.
–	0x3..0xF	Reserved for future extensions.

Local symbols have STB_LOCAL binding attribute and must be defined in the same object file in which the symbol entry appears. In other words, the symbol definition and all its uses must belong to the same compilation unit (part of a program that must be compiled as an atomic entity). The definition of a local symbol does not interfere with other symbols with the same name defined in other compilation units. Therefore, several local symbol entries with the same name may be linked together.
Global symbols have STB_GLOBAL binding attribute and are visible to all compilation units that make up the TPEF file. Only one file must contain the definition of a global symbol. The symbol must be undefined in all other compilation units that contain references to it. The definition of a global symbol will satisfy all unresolved references to a global symbol with that name in other compilation units.
Symbols with STB_WEAK attribute are similar to global symbols. They differ in the following aspects:

1. Weak definitions do not cause a linkage error if a global definition exists. In that case, the global definition simply “overrides” the weak definition.
2. The linker does not extract definitions of weak symbols in archive files (libraries) to try to resolve undefined weak symbols. Unresolved weak symbols do not cause linkage error; they are assigned value zero.

The symbol type provides a general classification of the associated symbol table entry according to the following table:

Symbol Type	*Value*	*Description*
STT_NOTYPE	0x0	The symbol’s type is not specified.
STT_DATA	0x1	The symbol is associated with a data object.
STT_CODE	0x2	The symbol is associated with executable code.
STT_SECTION	0x3	The symbol is associated with a section.
STT_FILE	0x4	The name of this symbol gives the name of the source file associated with this object file (possibly, the entire TPEF file).
STT_PROCEDURE	0x5	The symbol is associated with executable code and marks procedure name.
–	0x6..0xF	Reserved for future extensions.

Entries with type STT_DATA or STT_CODE refer to a program section. Their *st_value* member represents a section offset. An STT_DATA symbol table entry refers, for example, to variables, arrays or data structures. An STT_CODE symbol

table entry refers, for example, to function entry points or code labels. Entries with STT_SECTION type are related to a given section and typically contain relocation information. Entries with STT_FILE denote the point in the symbol section where the area dedicated to a given object file starts, and have always STB_LOCAL binding attribute. An STT_PROCEDURE symbol is similar to STT_CODE symbol, but STT_SECTION is used for indicating procedure start positions.
The meaning of members *st_value* and *st_size* for a given section type is given in the following tables:

Symbol Type	***st_value***	***st_size***
STT_DATA	Section offset.	Size of data object, in bytes.
STT_CODE	Section offset.	Unused.
STT_NOTYPE	Unused (zero).	Unused.
STT_SECTION	Section offset.	Size (byte)?
STT_FILE	Section offset? Unused?	Unused?
STT_PROCEDURE	Section offset.	Unused.

Typically, symbols are defined in relation to some section. A symbol's value must be updated (relocated) if the section is modified. The member *st_section* of a symbol table entry specifies the section for which the symbol is defined. If no section applies or if the section is undefined, then *st_section* contains the Null Section identifier (zero). The Null Section is described in Section 9 of this document.
Certain symbols, such as symbols that represent source file names (STT_FILE symbol type), have an absolute value. These values do not represent section offsets or addresses, and must not be relocated. Absolute symbols are signalled by flag *sto_abs* of member *st_other*, as shown in following table:

st_other	***Bit Mask***	***Description***
sto_abs	0x80	Symbol relocation type: absolute (1), relocating (0).
–	0x7F	Reserved to future extensions, must be zero.

Symbol ordering. [discuss] In each symbol table section, all local symbols precede weak and global symbols. The *sh_info* member of the symbol table section header holds the index of the first non-local symbol of the table. If present, a STT_FILE symbol table entry must precede all other entries that refer to the same file.

15.1. Symbol entries

Reserved entry for undefined symbol references. The first symbol table entry is reserved and has a special meaning. Its index (zero) is used in lieu of a proper symbol table index whenever the symbol referenced is unknown or no symbol is applicable. It contains the following values:

Field	***Value***	***Notes***
st_name	0	No name assigned.

Field	*Value*	*Notes*
st_value	0	Value is undefined: always zero.
st_size	0	Size of the symbol is not defined.
st_info	0	Symbol's type undefined.
st_other	0x80	Not a relocating symbol (value always zero).
st_section	0	Null section (section undefined).

See Section 25.6 for a possible extension of the symbol table entry format.

Symbol table entry for program entry point. To mark the first instruction of the program that is executed, TPEF provides a special symbol table entry with reserved name. The symbol table entry has the following values:

Field	*Value*	*Notes*
st_name	–	Assigned to string '`__tta_program_entry`'.
st_value	–	Section offset to entry instruction element.
st_size	0	Size of the symbol is not defined.
st_info	0x12	Code symbol with global binding.
st_other	0x00	Relocating symbol.
st_section	–	Code section that contains the program entry point.

16. Relocation section

Relocation is the process of connecting references to symbols with definition of the corresponding symbols. Relocation sections (*sh_type* SHT_RELOC) contain information that describes how to modify the references (addresses) contained in program sections.[5]

A relocation table applies to references of one section only. The first 2 bytes of the *sh_info* field of a relocation section header contain the identifier of the section that contains the references.

The format of the relocation section depends on the target architecture template (*fh_arch*) and is out of the scope of general TPEF specification. The reminder of this document section specifies the format of the target architecture template for *fh_arch* = TTA_TUT.

Field	***Bytes***	***Description***
r_offset	4	The section offset to the location where relocation applies.
–	1	Reserved for future extensions. Must be zero.
r_symbol	3	Symbol table index of the referenced symbol.
r_type	1	Type of relocation to apply.
r_asp	1	Address Space identifier of the address to relocate.
r_size	1	Bit width of the field containing the address.
r_bitpos	1	Bit offset into the address to relocate.

Member *r_offset* gives the location at which the relocation applies. This location contains a pointer to the symbol to be relocated. If the location belongs to a code section, the section offset points to the first byte of the immediate element that contains the address to be relocated.

Member *r_symbol* gives the symbol table index of the symbol with respect to which the relocation must be made.

Relocation chunks. In TTA programs, a pointer to a symbol may be cut into pieces called *chunks*. A chunk is a subword that, combined with the other chunks of the same pointer, forms the complete address of a symbol. A symbol pointer divided in chunks requires one relocation entry for each chunk.

Member *r_type* specifies the type of relocation to apply and how the relocation is divided in chunks. The upper half of *r_type* gives chunk information. The lower half gives the relocation action.

[5] Only *initialised program* sections can be relocated. Unitialised data (usually automatically zero-ed at startup) cannot contain variables containing zero as explicit initialization values, because such variables could represent the zero address of a program section, and thus require relocation.

The supported relocation actions are listed in the following table.

Value	*Encoding*	*Description*
RT_NOREL	0x0	No relocation.
RT_SELF	0x1	Absolute address, relocate relative to address self.
RT_PAGE	0x2	Paged address, relocate page offset.
RT_PCREL	0x3	PC-relative, relocate only if displacement changes.
–	0x4..0x7	Reserved for future extensions.
–	0x8..0xF	Reserved to specific architecture templates.

Relocation actions adjust a location that contains the address of another location (the target address). The value of the target address is stored in the section that contains the location of the reference.

The relocation action RT_SELF applies to absolute addresses. The relocator computes and adds a displacement to the original target address. Target address and the address of the location that contains the reference may belong to different address spaces.

The relocation action RT_PAGE applies to absolute addresses, but the location to be adjusted contains only the lower bits (the page offset) of the target address. The target address and the location to be relocated are in the same address space. The address refers to the memory page of the location where the reference is *made*. However, the location pointer at by *r_offset* is the location where the reference is *stored*. This location does not affect the displacement, as long as the distance between the two locations does not change.

The relocation action RT_PCREL applies to addresses that are stored as a displacement from the location of the reference to the target address. These addresses do not need adjustment (see section 25.8 for a possible extension to this type of relocation entries). Target address and the address of the location that contains the reference are in the same address space.

The address of the symbol to relocate can be divided in up to 8 chunks. Only addresses stored in code sections can be divided in chunks. A chunk may be encoded in one or more instruction slots of a long immediate or may be an in-line immediate.

The upper half of member *r_type* specifies whether the relocation entry is a chunk:

st_type field	*Mask*	*Description*
stf_chunk	0x80	Relocation applied to chunk (1) or complete address (0).
–	0x70	Not used, the bits must be set to zero.

If flag *stf_chunk* is set, then the relocation entry refers to a chunk of a larger address.

Member *r_size* gives the bit width of the address (or the chunk thereof) to relocate. The bit width is a number in the range [1,64] encoded with bias –1, that is, 1 has to be added to the value stored in *r_size* to obtain the actual bit width. The most significant bits (bit mask 0xC0) must be always zero.

Member *r_bitpos* gives the position of the least significant bit of the chunk into the complete target address word. In non-chunked relocation entries, *r_bitpos* is always zero. The most significant bits (bit mask 0xC0) must be always zero.

How relocation entries of chunked addresses are stored. The relocation entries of chunks that belong to the same address must be stored contiguously. It is an error if an unrelated relocation entry is stored between two relocation entries for chunks of the same symbol reference. Given that chunks must not overlap and must not leave unspecified bits, it is always possible to tell whether the group of relocation entries for a chunked symbol reference is complete. The last chunk has *r_bitpos*=0.

Example: Encoding of a group of relocation entries for a symbol address (original value: 0xA1B2C3) that is stored in three separate in-line immediates, 0x286, 0xB2, 0xC3, as shown in the following piece of code:

```
0x286 -> shl.1, 14 -> shl.2;
shl.3 -> r2, 0x32 -> shl.1, 8 -> shl.2;
shl.3 -> ior.1, r2 -> ior.2;
ior.r -> ior.1, 0xC3 -> ior.2;
ior.r -> ld.t;
```

Assuming that the in-line immediates 0x286, 0x32, 0xC3 are, respectively, 10, 6 and 8-bit long, three relocation entries are required for this reference to the symbol. The value of members *r_type*, *r_size*, *r_bitpos* of these three immediates is:

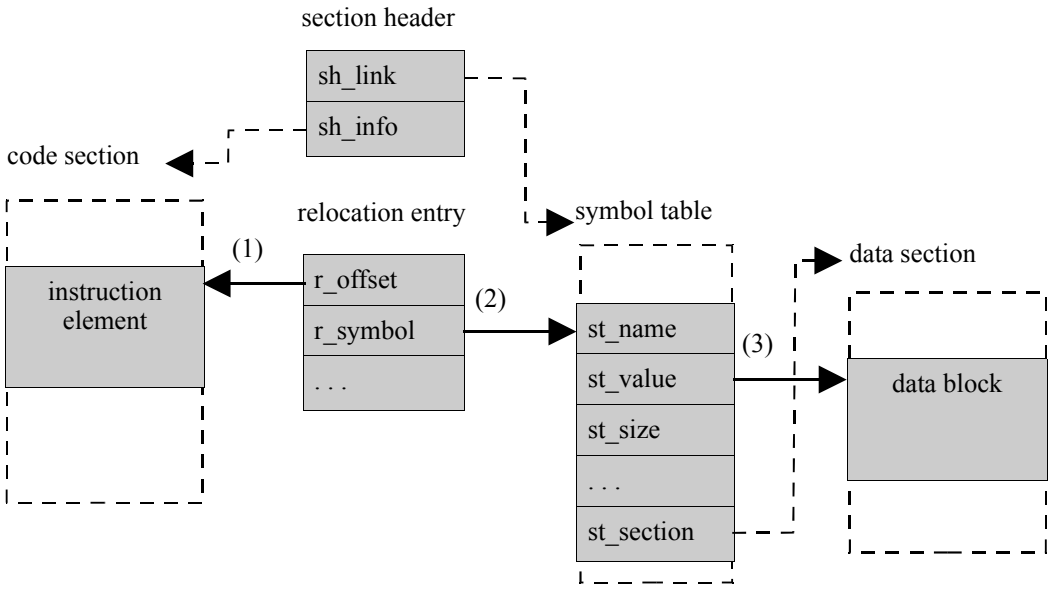
	r_type	r_size	r_bitpos
chunk 1	10000001	00001010	00001110

	r_type	r_size	r_bitpos
chunk 2	10000001	00000110	00001000

	r_type	r_size	r_bitpos
chunk 3	10000001	00001000	00000000

See Section 25.7 for possible extensions to the format of the relocation section.

The following diagram depicts the relations between a relocation section, a symbol section, a program section containing the reference to be relocated and the data section where the referenced symbol is stored.



The dashed arrows represent section references through *sh_id*. The solid arrows represent the following references between section elements of different sections:

1. Section offset to the first byte of the section element to which relocation applies.
2. Index of the symbol table entry for the symbol to which relocation applies.
3. Section offset to the first byte of the data block where the initialisation value of the referenced symbol is stored. If the symbol is not initialised, the data area is not (usually) stored in TPEF, and the section offset is the byte offset from the beginning of the uninitialised memory area, considering the conversion between MAU's of the address space and TPEF bytes described in Section 22.

Moreover, the size of the data block where the referenced symbol is stored is given by the symbol entry member *st_size*. This size is computed in bytes. For example, if the symbol data block takes 4 MAU's and each MAU of its address space takes 2 (8-bit) bytes in the TPEF file, then the data block size will be 8 bytes.

17. Line number section

Line number sections (*sh_type* SHT_LINENO) contain auxiliary information that can be used by symbolic debuggers to debug code at the source level. When present, the line number section contains a *line number entry* for every source line that can have a symbolic debugger breakpoint.
The format of the line number entries and the structure of the line number section are debugger-dependent.
The following format is proposed for the implementation of the TTA Design Framework, TUT.
The line numbers are grouped by function and are relative to a given function (this format is taken from COFF standard).

Fields	***Bytes***	***Description***
offset	4	Either section offset into the first instruction of the line, or the index of the symbol for a procedure symbol.
line_num	2	Line number, zero if this entry represents a procedure.

The first 2 bytes of member *sh_info* of a line number section contain the identifier of the section (of type SHT_CODE) the offsets of line number entries refer to.

[open issues: how to load/track the source code, how to locate procedure entries within it]

18. String Table section

String table sections (sh_type SHT_STRTAB) hold null-terminated character sequences, commonly called strings. The TPEF file uses these strings to represent symbol and section names and other section-specific information. Strings are referenced to as indices into the string table section.

The first byte, which is index zero, is defined to hold a null character (ASCII code 0x00). The last byte of a string table is defined to hold a null character, ensuring proper termination for all strings.

A string whose index is zero specifies either no name or a null name, depending on the context. An empty string table section is permitted; the *sh_size* member of its section header contains zero. Non-zero indexes are invalid for an empty string table. (Adapted from ELF specification.)

A string table index may refer to any byte in the section. A string may appear more than once; references to substrings may exist; and a single string may be referenced multiple times. Unreferenced strings also are allowed.

19. Debug section

The format of debug sections (*sh_type* SHT_DEBUG) is out of the scope of this specification and is dependent on the symbolic debugger used.

20. Machine Description File section

The optional machine description file section (*sh_type* SHT_MD) contains the specification of the target processor architecture that is expected to run the TTA program described in the file.

At most one Machine Description File section can be present.

The format of the MDF section is described in “MDF – Machine Description File Format for a New TTA Design Framework” document ().

When the MDF section is not present, the path name to the machine description file is expected to be stored in a special symbol table entry.

The machine description is needed to exactly reconstruct the parallel TTA program; the information stored in a TPEF file is *incomplete*.

[discuss: how to deal with mismatch between machine resource table and MDF]

21. Program Profile section

Program profile sections contain the profiles of a program run. The profile data consists of a table of fixed-size entries. Each entry contains an *execution count* and specifies how many times a given program element has been repeated during the program run.
A code section may have several independent profile sections, each containing the profile data of a different run of the program.
There are two types of execution counts:

1. Procedure execution counts.
2. Basic block execution counts.

The first part of the section data area contains all procedure execution counts, and is followed by all the basic block execution counts. Both procedures execution counts and block execution counts are stored in the same order in which they are laid out in the memory image.
For each procedure contained in the program, *n* execution counts are stored. The number *n* is fixed and depends on the latency of the transport pipeline of the target processor architecture. For details on this parameter, see the MDF specification document [1]. The execution count number 1 refers to the first instruction of the procedure (also called the procedure entry point); the execution count number 2 refers to the second instruction, and so on up to instruction number *n* from the procedure entry point.
For each basic block contained in the program one execution count is stored.
Format of execution counts. An execution count is a 64-bit integer word. All bits except the most significant are used to represent a 63-bit unsigned integer number in base 2. The most significant bit of the execution count must be ignored. Its use is reserved for future extensions.
Consistency checks. The minimal consistency checks that a TEPF profile section reader should perform are listed below:

1. The number of execution counts per procedure, *n*, must match the transport pipeline latency of the target processor. In case of unscheduled code, the transport pipeline latency is set to 1.
2. The number of procedure execution counts must be *n* times the number of procedures in the TTA program.
3. The number of basic block execution counts must be equal to the number of basic blocks that make up the TTA program.

22. Data Encoding and Alignment Rules

All data contained in the file, be it file structure, auxiliary information or program information, is stored in *big endian* format. The format in which the TTA programs are stored is independent from the data encoding of the target processor. Even the initialisation data need not reflect the actual bit image of the data in the target processor.

Simple alignment rules are applied when the bit width of the minimum addressable word (*asp_mau*) or the natural word (*asp_*wsize) defined in an address space are not 8, respectively 32 bits. An *n*-bits wide minimum addressable word is represented with the smallest number of bytes *b* such that $8b \geq n$. A natural word consisting of *w n*-bits words is represented with *w* vectors of *b* bytes. Since all references in the program sections are required to be *n*-bit words addresses, a machine address *A* corresponds to the file byte offset (relative to the section base address) *b* (*A – base*).

This alignment rule breaks down completely if the data encoding departs from 2's complement and IEEE-754 floating-point formats. For example, a 32-bit (or less) custom floating-point format may be defined which requires many more bits to be represented without precision loss in the host architecture format. In this case, the assumption that *w* words, each 8*b* bits long, are sufficient to represent *n*-bit words on the target architecture may not hold. [use symbol table entries to solve this problem?]

Thus, accessing data words using alignment rules may be the only option left, if symbolic information is not available, but should not be relied upon [force symbolic information to be present?]. The correct way to access any section element corresponding to an entity in the program, be it a data word or an instruction (field), is to lookup the symbol table entry (see Symbol Table section) corresponding to the machine address and retrieve the correct file offset to the referred entity.

23. Unscheduled Target TTA

Unscheduled (sequential) TTA code is not different from parallel TTA code in any fundamental way. Also the unscheduled TTA code depends on a target processor architecture. However, while the target architecture of scheduled code is specified by an ADF, the target architecture of unscheduled code is unique and is implied.

The target architecture of unscheduled code used by TTA_TUT architecture template is characterised by:

1. One transport bus (resource element 0x0000 of type MRT_BUS). The bus identification number zero is reserved to a special bus that identifies moves for which no physical bus is actually assigned.
2. One "universal" function unit that can perform all operations[6] in the program and contains all special registers (resource element 0x0000 of type MRT_UNIT). The unit identifier zero marks moves to operands and from results of operations that have not been assigned to a function unit fot the target processor.
3. One register file for integer variables (resource element 0x0080 of type MRT_RF). Register file identifier 0x80 is reserved to a special, unbounded register file that represents a pool of integer variables of predefined bit width.
4. One register file for Boolean variables (resource element 0x0081 of type MRT_RF). Register file identifier 0x81 is reserved to a special, unbounded register file that represents a pool of Boolean, 1-bit variables.
5. One register file for floating-point variables (resource element 0x0082 of type MRT_RF). Register file identifier 0x82 is reserved to a special, unbounded register file that represents a pool of floating-point variables of predefined bit width.
6. No long immediate registers: immediates values (of any bit width) are encoded in move source fields and are identified by resource element 0x0000 of type MRT_IMM. The immediate unit identifier zero does not represent an immediate unit. It signals that the immediate is in-line, that is, encoded in the source field of the same move that reads the immediate.
7. No bridge registers.

The exact specification of what is a "well-formed" sequential TTA may be subject to additional constraints. Such constraints, required by the tools that perform code analysis and parallel code generation, are not part of TPEF specification. For example, it may be convenient to restrict predication to jump operations and to admit only the Boolean register as guard term.

23.1. TPEF Move Format

In unscheduled moves, the source field is interpreted in a slightly different manner:

- Source general-purpose registers are actually program variables that represent

[6] With a one-cycle latency between trigger and result move.

temporary values.

- Output ports of the unique function unit are actually results of an operation.
- Immediate identifiers always represent in-line constants (immediate), never an immediate register (encoded by an instruction template).

The destination field of unscheduled moves is interpreted along the same lines of the source field (the destination function unit ports actually represent *operation inputs*).

23.2. Address Spaces

The target memory system of the sequential TTA code is characterised by two independent address spaces. One address space is dedicated to program code; the other is dedicated to all program data (initialised, not initialised, stack allocated or dynamically allocated at run time). The address spaces are typically (but not necessarily) characterised by the following parameters:

- start address: 0
- MAU: 8 bits
- natural word length: 4 MAU's (data), 1 MAU (code)
- word alignment: 4 MAU's (data), 1 MAU (code)

The choice of 8-bit MAU reflects the address space characteristics usually assumed by existing compilers. The instruction word length usually expected is 8 MAU's and reflects the size of sequential TTA moves generated by an existing frontend compiler.

Appendix

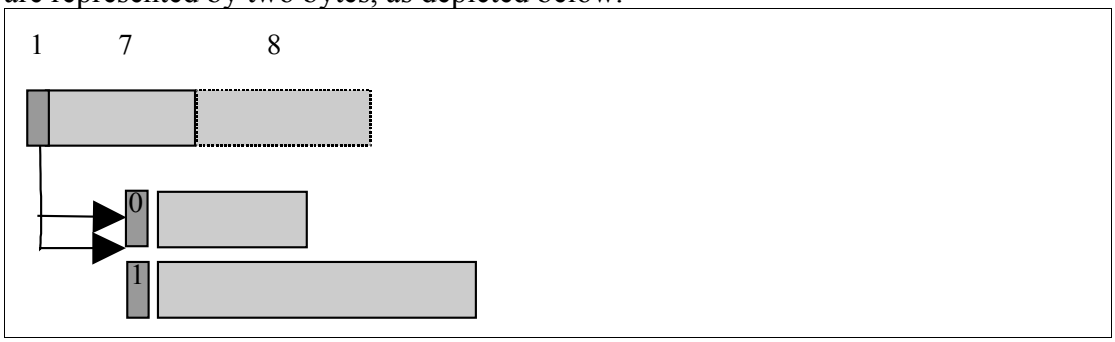
24. Glossary

FU	Functional Unit
MAU	Minimum Addressable Unit (of an address space)
ADF	Architecture Description File
ADFF	Architecture Description File Format
RF	Register File
TPEF	TTA Program Exchange Format
TTA	Transport Triggered Architecture
TUT	Tampere University of Technology

25. Further Ideas

25.1. Extensible Fields

An extensible field is a variable-length field that can represent a number in the unsigned range 0-32767. An extensible field can be one or two bytes long. Numbers in the range 0-127 are represented by a single byte, numbers in the range 128-32767 are represented by two bytes, as depicted below.



25.2. Immediate Sections

Introduce an *immediate section* where all immediates used by the program are stored. This section could be used by some instruction encoder/compressor, that store immediates in a small local memory and avoid direct encoding in the instruction stream.

25.3. Environment Section

Introduce a section to store information useful to define and setup the environment in which the TTA program is expected to run. Such information includes, for example, how to initialise the stack pointer register.

25.4. Address Space Reference Section

(17.03.2004)
Some source or destination moves represent values of computed (nonconstant) memory addresses. For correct interpretation of a TTA program that uses several address spaces, the address space of such nonconstant addresses must be somehow computed and stored permanently within the TPEF file. Unlike symbols that refer to constant addresses, computed addresses do not get any symbol table entry or relocation entry.
Introduce section to store address space information referring to move sources/destinations. A pre-scheduling pass must compute this information by propagating the constant terms that are at the bottom of the expression of a computed address.

25.5. Multicasting Support

(17.30.2004)
Extend the source id space of moves with a reserved identifier for multicasting (it means: continues to use the same source of the previous instruction element). By combining one move with normal source and several moves with this reserved id it is possible to define a transport from one source to multiple destinations.

25.6. Symbol Table Entry Extension

(19.04.2004)

In order to speedup linkage and processing of linked TEPF files, every symbol table could allow for two meanings of the *st_value* member of relocatable symbols:

1. In object files (with unresolved references), it represents the section offset relative to the section given by member *st_section*.
2. In linked files (without unresolved references), it represents the actual address where the symbol is located in the address space. The address space is still implied by the section of the symbol, given by member *st_section*.

The first encoding is convenient when it is necessary to relocate sections (that is, start address, given by *sh_addr* is changed) without modifying them. In this case, no relocation of symbol's value is necessary.

The second encoding is convenient when the section that contains the symbol is combined with another section (and thus destroyed), but the memory addresses of the section elements are unchanged. However, in this case it may be necessary to update member *st_section* of each symbol.

Member of *st_section*, when the second encoding is used, could contain the address space identifier instead of the section. This saves from the task of updating *st_section* member when a section is merged with another, but has the inconvenience of using one more member for two completely different purposes.

If the meaning of member *st_section* is maintained, then another type of inconvenience is introduced: the section of the symbol may be correct, but the address of the symbol stored in the *st_value* may be out of range.

A bit flag of member *st_other* could signal the type of symbol encoding.

25.7. Relocation Section with Addend Member

Certain processor architectures and object formats support relocation entries with one additional member, usually termed *addend*. This member contains the value that requires relocation. Usually, when the value is stored in the addend, the location that contains the pointer contains zeroes.

The main advantage of relocation entries with addend member is that all the terms needed to compute the relocation are available in the relocation entry. Access to the section being relocated becomes unnecessary. In case of chunked addresses, a further advantage is that the entire value to be relocated is stored in the addend of every

relocation entry for that chunked symbol reference. This reduces the computations needed to relocate a chunked address, since bit carry or borrow that can occur between chunks is handled automatically.

The main disadvantage of relocation entries is that they limit the bit width of the addresses to be relocated to the (fixed) size of the addend, which for practical reasons should be smaller than the maximum immediate size (16 bytes). Typically, the addend is 4 bytes long in object formats where it is present.

25.8. Support for PC-Relative Relocation

On traditional linkable or object file formats, PC-relative symbol references are treated as nonrelocatable addresses. However, this limits the use of PC-relative addressing in the program code to compile-time resolved addresses. If addresses that may be resolved at link time were admitted, then also PC-relative addresses would need adjustment. PC-relative relocation would be required only if the displacement between the location of the reference and the target address changes.

Relocation of PC-relative references poses a problem. The displacement is computed with respect of the address of the location where the reference is *made*. However, the location pointer at by *r_offset* is the location where the reference is *stored*. This location does not affect the displacement.

Example: Given the following code:

```
0x100: . . . [d -> im1]
0x108: . . .
0x110: im1 -> jump
```

A displacement *d* is stored in location 0x100 as an immediate, but the immediate value is used by an operation located at 0x110. The target address is computed as 0x110 + *d*, not as 0x100 + *d*. However, the relocation entry *r_offset* member gives the location 0x100, where the target address is encoded, because that is where the relocation adjustment takes place.

A possible solution to this problem is to provide an additional, contiguous relocation entry that specifies the location of the PC-relative reference. This entry could be identified by member *r_size*=0 and could either follow or precede the other relocation entries of the symbol reference.

26. Notes

The following notes are not strictly part of the TTA Program Exchange Format specification, but give useful insights on the problems and implications of the design decisions taken.

Information stored to represent TTA instructions. The information contained in a TPEF file can accurately describe the visible state and the behavior of parallel TTA programs, not the precise architecture on which it runs. In other words, the information stored in the file does not need to completely specify which target architecture resources are assigned to every instruction, move, or immediate. (For one thing, redundant specifications can be left out of the file without reducing the accuracy of the program representation.) In practice, the move member does not contain precise information on the transport network connections used to route the data transport specified by moves.

Resource assignment need not be specified explicitly when, given the target processor architecture for which the TTA program was compiled, the assignment of resources that are not specified in the TPEF file is "trivial". Trivial hear means that either the assignment is unique (thus no information is needed) or that the choice is irrelevant. The latter case occurs for *orthogonal* target architecture resources. [discuss: Such *is not* the case is some cases, for example when the connectivity of the transport network is restricted.]

An implication of this aspect of the TTA-PE format is that a TTA parallel program could constitute a valid program for target processors other than the original target for which the code was scheduled. In other words, a parallel TTA program stored in a TPEF file does not dictate unambiguously its target processor architecture, but rather drastically restricts the set of target processor architectures that can run it.

Interpretation of relocation offset. In other executable/object formats, such as ELF, the interpretation of the *r_offset* field of a relocation entry is "virtual address". Using virtual addresses to refer to relocation sites of a program section is very problematic for TPEF code sections, because the instructions (CODE section entries) are stored using a variable number of bytes. Locating an instruction given its address in the memory image requires scanning the entire code section. Storing section (byte) offsets instead is much more efficient.

27. Resolved Design Issues

During the design of the TTA Program Exchange Format, many issues had to be confronted. Most issues required several design alternatives to be considered. This section details the reasons that motivated the design decisions taken.

1. *Use of file offsets instead of (virtual) addresses to refer to entities contained in a program section.* Some binary formats (COFF, for example) store references to program entities --such as instructions (operands) and data words-- as virtual addresses. These addresses are expressed relative to the base address of the section that contains the entity and represent its address in the memory. The use of virtual addresses is not easily applicable to TPEF files. [CHECK: First of all, the entries of some program sections may be stored using a variable size encoding.] Second, the information in a section entity may be stored in a form that does not reflect the actual memory layout on the target machine. For these reasons, there is no simple linear relation between entry addresses in memory and file offsets into the program section. File offsets, on the other hand, are unambiguous and are not affected by the binary encoding and the data size of the program elements nor by the form in which program elements are stored in the file. These reasons justify the use file offsets to refer to any element in a program section.
2. *Encoding relocation information in separate section instead than as part of each program section*. Some binary formats (notably COFF) store a table of relocation entries for a given program section at the end of the section itself. Other formats, like ELF, store the relocation table for a given program section into a separate relocation section. By keeping a separate section for each relocation table: (1) information is kept clearly separated, tools that do not use relocation do not need to be aware of the relocation table, (2) the format of the single sections is simpler, (3) it is possible to strip relocation information without having to modify or even analyse the corresponding program section.
3. *Use of explicit size fields for every data structure of the file.* The size of every data structure specified in TTA-PE file format is explicitly declared in dedicated fields. The file format identification structure, the file header and the section headers, for example, all contain such fields. The size of a structure may be declared in a size field of the structure itself or in a size field of a structure at a higher hierarchical level. The size of a section element, for example, is not contained in each element, since this would be wasteful, but in the section that contains the element. Notice that size is not applicable to variable-length data structures. In this case, the size field is ignored. By referring to the declared sizes to locate the fields in the data structures, compatibility with future extensions is assured, on condition that the extensions do not affect the interpretation of existing data structures. This design decision and its motivation are inspired by the ELF file format.
4. *Choosing a target architecture for sequential TTA programs that fully fits the architecture template of the target processors.* By clearly defining the target architecture of the sequential code in terms of the target processor templates, parallel and unscheduled TTA code can be conveniently represented using the same data format, while at the same time the data format is kept simple and regular.
5. *Reserving identification numbers of processor components to the target architecture for sequential TTA programs.* By reserving a special identification number to any processor component that can be referred to (as assigned hardware resource) in a move, it is possible to store and easily difference unscheduled and parallel code within a TTA program. It is also possible to represent partially scheduled moves, whereby some hardware resources have been assigned, while others haven't.
6. *Avoid use of bit fields.* For portability reasons, bit fields are not used in TPEF. The values of all flag combinations are explicitly indicated as numbers. The MSB of an 8-bit byte, for example, is indicated by "0x80", and should be tested by masking with the such number.
7. *Relocation entries contain a pointer to the first byte of the element that contains the address to be relocated, rather than a pointer to the (first byte) of the raw data representing the address.* This choice is in contrast to that of popular object and link formats (a.out, ELF, COFF). In TPEF, this choice is preferable because (1) it is safer (more checks that the element containing the reference is well formed) and (2) works also with variable encoding of the preceding bits of data.

8. *Invalid (null) section, address space.* The mandatory invalid section of type SHT_NULL and the Invalid Address Space identifier zero parallel the concept of *Null Object* common in object oriented design community. It also matches an ELF design criterion.
9. *Unique section identification numbers across entire file.* The use of a unique number (rather than, for example, a unique number for a program section and all auxiliary sections referring to it) has practical motivations: it should make lookup of individual sections easier, since it does not require to look into the *sh_type* flags to tell one section from another. A possible disadvantage is that an additional field is required in each auxiliary program section, to define the referred program section.
10. *Single Architecture Description File.* A TPEF file can contain program data for one target architecture only. Therefore, there can be at most one machine resource table and one architecture description file section.
11. *Global address space table section.* An alternative would be to have one address space section for each address space. In this case, the section identifier of the address space section could be used. This alternative was rejected because it would mix section ids with address space ids.
12. *Reserved data items for undefined references.* To facilitate development of robust implementations of TPEF, whenever a reference to a given type of data items may be undefined or not applicable, a special instance of the data type must be reserved for such references. This enables implementation of such reference as a pointer or reference to a well-formed (albeit, special) data structure, rather than forcing a NULL pointer or an illegal value. The object-oriented equivalent to this guideline is the *null object* design pattern.

1 A. Cilio, H. Schot, and J. Janssen: “Machine Definition Format for a New TTA Design Framework”, S-003.
2 A. Cilio: “Processor Architecture Specification for TTA Codesign Environment”.
3 A. Cilio: “TCE Architecture Template Programming Interface Fuctional Requirements”, S-010.