The teq Processing Flow

version 1.0, Sunday, October 22, 2017

Kiva Design Groupe pete wilson **M** 512 577-1618 pete@kivadesigngroupe.com

Table of Contents

The teq Processing Flow

Pete Wilson v0.3 October 7 2019

Matches teq 0.7v3.1.1[October 7 2019]

This document describes how the teq architecture and program compiler functions, concentrating on the very high level aspects.

Changes from prior version:

*The most significant change is that things like variables, registers, fields etc, all of which need a Symbol to hold their name, no longer have an 'embedded' Symbol in their definition. Rather, the Symbol has one new field, a void * ptr, which points at a separate structure holding relevant information; and that structure has a pointer back to its Symbol. We provide new functions to get the separate structure from a Symbol; this checks the Symbol has the appropriate symclass and subclass. This change was done to help avoid stupid programming errors in treating one sort of structure as another, causing havoc in the symbol tables (when a smallish entity was mistakenly updates as if it were a larger one, bad things happened to memory-adjacent symbols)*

Overview:

The general structure is:

- an architecture description parser, handwritten, for the architecture resources, along with a program parser (most of which we wish to re-use for parsing teq programs) which parses the 'action' definitions - which are just teq program, with the complication that we use the <- operator to mean 'read or write an architecture element (like a register)' rather than 'send or receive a message', as it would be in teq-for-programs

The handwritten program parser works like this:

- it parses the source of the action definition

- using a token (the symbols which are read from the source) symbol table; each Symbol has classification info based on its properties as a Token (a number, a string, a name...)

- with an operator-precedence expression parser, which generates simple trees whose elements are operator plus one or two operands held in that token symbol table. Errors detected are limited to malformed expressions. When reducing an expression, the parser generates new Symbols, which are marked as temporaries. We do not know the type of a temporary at this stage; it must be calculated later. The temps are added to the token symbol table.

A Symbol now looks like this:

```
typedef struct {
    qBlock header;       // next struct in the hashchain
    char * name;         // pointer to name
    uint32 prehash;      // the prehash - results of basic hashing
```

```
                     // computation, invariant across symtab sizes
    int32 number;        // we will want an intermediate form of triples
                         // in which the fields are symbol ***indices***
                         // not pointers.
    void * ptr;          // NEW - points at relevant extra data
    TokClass tokclass;
    symClass symclass;       // what sort of symbol is it?
    union {
      symSubClass subclass;  // for when we have a class/subclass
                             //relationship
      uint32 index;          // for when we want simply to count
                             //things, as in a macro argument
      } sub;
} Symbol;
```

- errors detected in the expression parser are parseError() parseValError()(errors detected directly in the source), (errors detected in processing, like lacking a name or an operator in internal stacks) and badError()(implementation errors, like looking for the name of an operator and finding there isn't any such name)

- throughout the program, we do a lot of dynamic memory allocation. It's not likely to fail, but the standard action if it does is to call allocError()

- the parser generates a list of *Statements* of various types. Statements are structures of various types. Simple statements,like an assignment statement, have a tree representing the expression parsed.Errors detected include parseErrors()s, like not seeing a '{' where expected in an if or a while.}

- complex declarations (like int32 fred, jim, bert = 0, alf;nt32 fred; int32 jim; int32 bert; bert = 0; int32 alf;) are converted to a sequence of very simple declarations and separate assignment statements (so that would be i). Statements represent the declarations.

- we represent the behaviour of code as a sequence of Triples. A Triple may be queued using the standard teq functions. It contains, most importantly, three pointers to Symbols in the raw symtab, and three pointers to Symbols (scoped, declared, checked) in the refined symtab queue. Basic parsing fills in the Symbol pointers.

A Triple now looks like this:

```
typedef struct {
    qBlock header;
    uint32 seq;
    uint32 triple_num;
    tripleType type;
    tripleStatus status;
    opSpec * op;

    // the operands for the initial node-to-triple pass, sans typechecking
    // etc these point to Symbols in the raw symtab
    Symbol * symd0;
    Symbol * syms0;
    Symbol * syms1;

    // the operands after conversion
    // these point to Symbols in the queue of symtabs
    Symbol * d0;
    Symbol * s0;
    Symbol * s1;
} Triple;
```

- the statements have their trees or other data structures pointed at by a void pointer to an appropriate structure (several, for example in a while statement there's a controlling expression tree and a sequence of body statements)

- a statement has a single triple queue

- a first Conversion pass through the Statements and their queues of Triples creates the d0, s0, s1 Symbols in each Triple; these are created by the conversion path. Declaration statements cause the relevant Symbol to be added to the scoped symtab structure. The refined structure is just a queue of symbol tables; this is NOT created during parsing, but in a later stage - conversion. When we encounter a declaration statement, we introuce the typed variable into the symbol table. When we encounter a block {...}, we lex in, which creates a new symtab for the new scope; this is added to the front of the queue of symtabs. On exiting the block, we remove the symtab at the head of the queue.

We provide lookup functions for both the front of queue (to check for within-scope multiple declaration errors) and up the queue (to find names in outer scopes).

- Symbols representing declared variables have their ptr point at a varInfo structure, which includes a pointer to the Symbol representing its type plus various usage counters and info.

- the variables for temporaries are created and declared during the process of 'converting' raw triples into refined triples. We recognise a temp because its Symbol in the raw symtab is of symClassTemp. We use checkDeclareTemp() in the conversion function to see if the destination of the operation is a temp. If it is, we create a variable for it and fill in info marking it as a variable and that it needs its type calculated (*symClassVar, symSubComputeType*). We then create a new statement declaring it and add that statement in immediately prior to the current statement.

- while converting the Triples to have variables, we look up the associated Symbol in the refined symbol structure. If the variable isn't found, we have an error (a statementError()).

- once we've built the refined Triples, we revisit it and run back down it checking for errors (type errors, operation errors,..). teq requires the operands in an expression be of the same type. We provide type casts of the go form: to make fred, which is an int32 into an int 16 we write `int16 shortfred = int16(fred)` - that is, the typecast looks like a function call, not like the C form.

- for each triple, we check the variables exist and see if any are of class *symClassVar:symSubComputeType*, subclass. If so, we compute the type of the Var from the types of the source operand(s).

The end result is an architecture specification which has a hierarchical definition of the encoding of instructions, with data structures representing the architecture (resources and fields), with each instruction having a program representation consisting of a queue of Statements, each with their Triple queues (etc).

To create a model, we walk the specification, writing a set of .h and .c files which can be linked to pre-written C code to construct an interpreter and an assembler. For the interpreter body, we use nested case statements to implement decode, and use a *printtripleAsC()* function to translate statements into C.

Implementations

This version of teq is able to parse and capture an implementation specification. This allows the declaration of function units in a processor implementation, along with

- the latency and repeat numbers for the function units
- which instructions use which function units.

A simple implementation for an accumulator machine, acc32 is:

```
implementation fred {
    contexts 4, priorities 8;
    funcunits arith_unit, mul_unit, div_unit, ldstore_unit, shift_unit,
          branch_unit, nop_unit;
    arith_unit: latency = 1, repeat = 1;
    mul_unit: latency = 4, repeat = 1;
    div_unit: latency = 16, repeat = 16;
    ldstore_unit: latency = 2, repeat = 1;
    shift_unit: latency = 1, repeat = 1;
    branch_unit: latency = 2, repeat = 1;
    nop_unit:latency = 0, repeat = 0;
    arith_unit instructions:
       readx, ready, writex, writey, add, sub, and, or, xor, ldc;
    mul_unit instructions: mul;
    div_unit instructions: div;
    shift_unit instructions: shr, shl;
    ldstore_unit instructions: loadx, storex, loady, storey, loadxinc,
       storexinc, loadyinc, storeyinc, loadlocal, storelocal;
    branch_unit instructions: beq, bgt;
    nop_unit instructions: haltcpu;
}
```

The goal is to be able to generate time-tagged models; right now, the implementations are captured, but nothing is done with them (you can't generate a time-tagged model).

Further, we probably need to be able to describe a more complex microarchitecture; the current stuff assumes that registers cost no time to access, and an instruction uses just one unit.

Program

This version of teq can parse, check (somewhat) and translate teq **program** (rather than just architecture) into triples, and output those in a textual representation very close to the triples themselves; this representation is still in early days. The major structural change was the introduction of defining and calling functions; things work otherwise just like they did in handling the actions of instructions when handling architectures.

We hope to avoid the need for a linker by in essence combining arbitrarily many files at compile time. We hope to use the triples format to cache translations of files. The effect - if all works as hoped - is that we won't need header files (the compiler's multipass, and so we will see all of the program before needing to compile).

Here's a simple program created purely for testing:

```
func float32 sin(float32 x) {
float32 sine = x * 0.123 + (x * x * 0.234) + (x * x * x * 0.012);
return sine;
}
func float32 cos(float32 x) {
float32 cosine = x * 0.123 + (x * x * 0.234) + (x * x * x * 0.012);
return cosine;
}
func int32 splod(int32 a) {
float16 jim;
int16 fred;
fred = fred + int16(jim);
return a + a;
}
func void main(void) {
int32 x, y, z;
x = x * z;
x = splod(17);
float32 angle = 0.1234;
float32 p, q;
p = sin(angle);
q = cos(angle);
p = p * p + q * q;
}
```

Here’s a representation of that program in the current early version of the triples format:

```
func float32 sin args 1
     arg 0  float32 x
     float32 sine
     uint32 teq_temp_0
     uint32 teq_temp_1
     uint32 teq_temp_2
     uint32 teq_temp_3
     uint32 teq_temp_4
     uint32 teq_temp_5
     uint32 teq_temp_6
     uint32 teq_temp_7
     uint32 teq_temp_8
     op_mul teq_temp_0 x 0.123;
     op_mul teq_temp_1 x x;
     op_mul teq_temp_2 teq_temp_1 0.234;
     op_plus teq_temp_3 teq_temp_0 teq_temp_2;
     op_mul teq_temp_4 x x;
     op_mul teq_temp_5 teq_temp_4 x;
     op_mul teq_temp_6 teq_temp_5 0.012;
     op_plus teq_temp_7 teq_temp_3 teq_temp_6;
     op_assign NULL sine teq_temp_7;
     op_return NULL sine NULL;
     endfunc
func float32 cos args 1
     arg 0  float32 x
     float32 cosine
     uint32 teq_temp_9
     uint32 teq_temp_10
     uint32 teq_temp_11
     uint32 teq_temp_12
     uint32 teq_temp_13
     uint32 teq_temp_14
     uint32 teq_temp_15
     uint32 teq_temp_16
     uint32 teq_temp_17
     op_mul teq_temp_9 x 0.123;
     op_mul teq_temp_10 x x;
     op_mul teq_temp_11 teq_temp_10 0.234;
     op_plus teq_temp_12 teq_temp_9 teq_temp_11;
     op_mul teq_temp_13 x x;
     op_mul teq_temp_14 teq_temp_13 x;
     op_mul teq_temp_15 teq_temp_14 0.012;
     op_plus teq_temp_16 teq_temp_12 teq_temp_15;
     op_assign NULL cosine teq_temp_16;
     op_return NULL cosine NULL;
     endfunc
func int32 splod args 1
     arg 0  int32 a
     float16 jim
     int16 fred
     uint32 teq_temp_18
     uint32 teq_temp_19
     uint32 teq_temp_20
     int16 teq_temp_18
     op_convert teq_temp_18 jim int16;
     op_plus teq_temp_19 fred teq_temp_18;
     op_assign NULL fred teq_temp_19;
     uint32 teq_temp_21
     op_plus teq_temp_21 a a;
     op_return NULL teq_temp_21 NULL;
     endfunc
func void main args 0
     int32 x
     int32 y
     int32 z
     uint32 teq_temp_24
```

```
uint32 teq_temp_25
op_mul teq_temp_24 x z;
op_assign NULL x teq_temp_24;
uint32 teq_temp_26
int32 teq_temp_26
uint32 teq_temp_27
op_passarg 17 0 splod;
op_call teq_temp_26 splod NULL;
op_assign NULL x teq_temp_26;
float32 angle
uint32 teq_temp_28
op_assign NULL angle 0.1234;
float32 p
float32 q
uint32 teq_temp_30
float32 teq_temp_30
uint32 teq_temp_31
op_passarg angle 0 sin;
op_call teq_temp_30 sin NULL;
op_assign NULL p teq_temp_30;
uint32 teq_temp_32
float32 teq_temp_32
uint32 teq_temp_33
op_passarg angle 0 cos;
op_call teq_temp_32 cos NULL;
op_assign NULL q teq_temp_32;
uint32 teq_temp_34
uint32 teq_temp_35
uint32 teq_temp_36
uint32 teq_temp_37
op_mul teq_temp_34 p p;
op_mul teq_temp_35 q q;
op_plus teq_temp_36 teq_temp_34 teq_temp_35;
op_assign NULL p teq_temp_36;
endfunc
```

Futures:

- Looking at simplifying the parsing so that the parse phase uses a lexed-in symtab, which is lexed out from on completing parsing. This should save a number of error-prone double symtab installation operations.

- Need to implement real boolean expressions (with jumping code).

- Need to continue work on the triples format, in several ways

    - The textual format will need to turn the declarations into an array

- The triples themselves need to reference their symbols by an index number, not by name (to speed parsing); for human readability we can add a very-quick-to-lex comment of the triple using the names
- currently, each statement has its own triples queue; to make multi-statement optimizations practical we will want a 'flattened triples' internal representation, holding all triples on the same queue and turning the current representation of (eg) loops - which have identified expressions and body parts - into labels and branches. This should also simplify translation into target machine language, since it *should* ease the instruction selection (eg, when there are several ways to add a literal - a single instruction if the literal is small enough in value, an instruction sequence otherwise).