Tom's Eviscerated Processor Instruction set Design (Tepid)

Tom Murtagh (borrowing quite heavily from Duane A. Bailey)

December 3, 2012 (software version: 2.139)

This document describes Tom's Eviscerated Processor Instruction set Design (Tepid), a small subset of the Williams Academic Risc Machine (Warm) instruction set architecture (ISA) designed by Duane Bailey. The hypothetical Warm machine was motivated by the design of the ARM[1] family of processors, but has (often substantially) different functionality. Tepid is a strict subset of Warm. Tepid was derived from Warm by removing as many features as possible to simplify the process of implementing an interpreter for the architecture while maintaining the ability to express interesting programs. The Tepid ISA, like the instruction sets of many RISC-style machines, is a load/store architecture with fixed length instructions that make use of a limited number of addressing modes. Most instructions are conditionally executed and have variants that allow the setting of condition code bits.

1 Introduction

Tom's Eviscerated Processor Instruction set Design (Tepid) is a 32-bit RISC-style virtual processor that is loosely modeled after the ARM family of architectures.[2] The Tepid processor has a 32-bit datapath with a 24-bit address bus addressing 16 million words of store.

The Tepid instruction set includes a select group of instructions that make use of a small number of addressing modes. Each instruction is fully encoded in a 32-bit word. Because of these limitations, memory may only be accessed by a few 'load' or 'store' instructions, placing it squarely in the LOAD/STORE architecture design space.

RISC programs are frequently longer than their CISC counterparts. Part of the reason is the relatively low capability of traditional RISC instructions. In the Tepid architecture, this deficiency is counteracted in two fundamental ways: (1) most instructions are able to shift one operand with no overhead, and (2) all Tepid instructions can be conditionally executed, based on the current condition code bits. These two features can reduce the number of instructions in many hand tooled assembly language programs, considerably.

2 The Architecture

The Tepid machine is a 32-bit processor. All data manipulated by the machine is 32-bits (a 'word') long. Primitive types that require less space are typically stored in the least significant bits of a word. For example, while strings of ASCII characters could be packed four per word, Tepid programs typically store each character in a separate word-addressable memory location.

At the core of the processor is a set of 16 general purpose registers, `r0` through `r15` (see Figure 1). The `r15` register (also called `pc`) is always interpreted as the *program counter*—a pointer to the instruction currently being executed.[3] The `r14` register (or, alternatively, `lr`) is the *link register*. Typically this register contains the return address associated with the currently executing subroutine, but the user may use this

[1] ARM is a registered trademark of ARM Holdings.

[2] The Advanced RISC Machine line of architectures are quite popular targets for low-power devices including media players, cell phones, and solid state 'netbook' style portable computers.

[3] The original Warm documentation described the program counter as a pointer to the next instruction to be executed, but experimentation with the Warm simulator, wai, suggest that if r15 is reference in an instruction the value obtained is the instruction's own address rather than the address of the next instruction.

as a general purpose register if desired. The `r13` register (or `sp`) is the `stack pointer`. It always points to the last value added to a stack of values in memory. No instruction actually depends on this interpretation of this register, so it may be used as a general purpose register if desired. The remaining registers can be used in any way desired. For example, it is common to pass parameters to subroutines by placing them in lower numbered registers, and to return function values in register `r0`. When independently written programs must be linked together to function as a single unit, a *call standard* is typically agreed upon, but Tepid makes no assumptions about the particular approach used.

Register	**Alias**	**Typical Use**
`r0`	`a1`	argument 1 (caller saved) and return value
`r1`	`a2`	argument 2
`r2`	`a3`	argument 3
`r3`	`a4`	argument 4
`r4`	`v1`	local variable 1 (callee saved)
`r5`	`v2`	local variable 2 (callee saved)
`r6`	`v3`	local variable 3 (callee saved)
`r7`	`v4`	local variable 4 (callee saved)
`r8`	`v5`	local variable 5 (callee saved)
`r9`	`v6`	local variable 6 (callee saved)
`r10`	`v7`	local variable 7 (callee saved)
`r11`	`fp`	frame pointer (callee saved)
`r12`	–	O/S reserved register (avoid)
`r13`	`sp`	stack pointer
`r14`	`lr`	link register (callee saved)
`r15`	`pc`	program counter and ccr (top 4 bits)

Figure 1: The registers available to Tepid, and their common interpretations.

Tepid supports access to a 16 million word addressable store. As a result only the low 24-bits of a register are ever used to develop an effective address.

Three bits, named N, Z, and V, are set by many instructions to reflect that the result value was zero (Z) or negative (N), or that the last add-like operation generated a signed overflow (V). Because the most significant bits of the program counter are unused, the Warm architecture uses the upper bits of the program counter to store the current condition code bits. While these bits are also used in Tepid, implementations of Tepid are not required to store their values in the program counter. They may be kept in a separate, special purpose register. Figure 2 documents this organization.

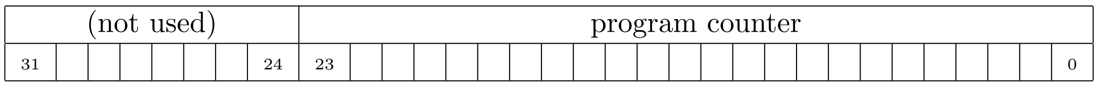

Figure 2: The interpretation of `r15`, or `pc`, the program counter.

Condition				Ending	Meaning
31	30	29	28		
0	0	0	–	–	always
0	0	1	–	`nv`	never
0	1	0	–	`eq`	equal (`Z`$= 1$)
0	1	1	–	`ne`	not equal (`Z`$= 0$)
1	0	0	–	`lt`	less than (`N`$\neq$`V`)
1	0	1	–	`le`	less or equal ((`Z`$= 1$) or (`N`$\neq$`V`))
1	1	0	–	`ge`	greater or equal (`N`$=$`V`)
1	1	1	–	`gt`	greater than ((`Z`$= 0$) and (`N`$=$`V`))
–	–	–	0	–	don't set the condition codes
–	–	–	1	`s`	set condition codes

Note: The ending `al` explicitly indicates the always relation.

Figure 3: The condition encodings.

3 Instruction Formats

The instructions of the machine are encoded in one of 2 basic formats: type 0 (or *arithmetic*), and type 1 (or *load/store*)[4].

3.1 Features Common to All Formats

Each format includes a condition field that controls the instruction's interaction with the condition code register. Nearly all instructions can be conditionally executed. By adding two letters after the operation code, the instruction executes if the current setting of the condition codes matches the specified condition. For example, the `addne` will perform an addition, but only if the last condition-setting operation generated a non-zero value.

If an additional s is added to the end of the operation code, the instruction will *set* the condition code bits, typically based on the result of the computation, or the shift involved. In Tepid unlike Warm, this option can only be applied to type 0 (arithmetic) instructions.

3.1.1 Register Direct Mode

In this mode, a register is involved as a source or destination for the computation. If a value is needed, it is found in the specified register. If the register appears as the first operand—the destination of the operation—the result will be stored in that register. The destination and the left hand source, if they are specified, are always in register direct mode.

Registers are represented by a four bit value that corresponds to their register number.

[4]The Warm architecture included an additional format for branch instructions. This format supported 24-bit offsets for branch destinations making it possible to branch to any location in memory. We cannot imagine anyone ever writing a Tepid program large enough to require a branch offset larger than supported by the load/store format. Therefore, in Tepid, an ADR instruction (described below) with the program counter register as its destination should be used in place of any Warm branch instruction.

3.2 Type 0: The Arithmetic Format

Most arithmetic instructions take two source operands, combine them using the arithmetic operation, and write the result to a destination register. One of operands, the left hand source, is always a register. The right hand source can be an immediate value, a register, or the result of applying a shift operation to a register. The encoding of the instruction bits is summarized in Figures 4 and 5.

Arithmetic instructions have a zero in bit 27.

condition				0	opcode				dest reg				src reg				source 2 operand														
31			28	27	26			23	22			19	18			15	14														0

Figure 4: The type 0 format: arithmetic instructions.

0	exponent					value								
1	0	0	shop		src reg 2				shift count					
14	13	12	11	10	9			6	5	4	3			0

Figure 5: The second arithmetic source formats.

3.2.1 Immediate Mode

Immediate mode (limited to use as the right-hand source) allows for the insertion of values into a computation that are derived from the instruction itself. In the assembly language, an immediate value is indicated by a hash mark (`#`) followed by a value. An integer value can be specified in decimal, octal (with a leading `0`), or hexadecimal (with a leading `0x`). The ASCII value associated with a character can often be specified using a leading apostrophe (`'`). When a symbol is used, its corresponding definition (typically a small integer or an address) is inserted.

Because there is limited space in the instruction to store an immediate value, it is stored as a 9-bit unsigned integer that is shifted by 0-31 bits. This immediately limits the type of values that may be used to those having no more than 9-consecutive non-zero bit values. The assembler will determine if the particular value can be stored, and will warn you if the value will not fit. Notice that most negative values cannot be specified directly in immediate addressing. The `mvn` instruction facilities the storing of values with a large number of ones.

This mode may be identified by a zero in bit 14.

Examples:

```
mov     r0,#3    ; store the value 3 in r0
mvn     r0,#0    ; store the value -1 in r0 (see mvn description for details)
```

3.2.2 Register Shifted by Immediate Mode

When the register appears in the right hand source, it is always shifted. In this mode, the register is shifted by an unsigned immediate value between 0 and 31 (written, for example, as `r1, lsl #1`)[5]. The type of

[5]Warning: The instruction format for such operands allows 6 rather than 5 bits for the shift amount. This makes it possible to encode shift amounts greater than 31 and the assembler will let you do this. In your implementation of TEPID, you should assume all shift amounts will be less than 32 and simply ignore the high-order bit.

Shop		As'y	
11	10	Code	Meaning
0	0	`lsl`	Logical shift left (see note)
0	1	`lsr`	Logical shift right
1	0	`asr`	Arithmetic shift right
1	1	`ror`	Rotate right

Note: `asl` is a synonym for `lsl` in the assembler.

Figure 6: Encoding of shift operations.

shift may be a logical or arithmetic shift in either direction (`lsl` or `asl`, `lsr`, or `asr`), or a right rotation (`ror`). The assembler allows an abbreviated form of this mode that appears and acts like register direct, but is encoded as a logical left shift by zero bits. These shift operations are encoded as in Figure 6.

This mode is identified by the the value 100 in bits 14 through 12.

Examples:

```
adds    r0, r1, r1, lsl #1; r0 is r1*3, conditions are set based on the addition
movs    r0, r1, asr #3    ; r0 is r1/8. Condition is based on shift
movs    r0, r0, ror #1    ; the circular queue is rotated one bit
add     r0,r1,r2          ; the r2 is encoded as r2, lsl #0
```

3.3 Type 1: The Load/Store Format

Access to memory is limited to a small collection of instructions whose sole purpose is to fetch and store register values. As a result, the addressing modes available for load/store operations reflect the typical ways that registers are transferred to memory. The format of these instructions is detailed in Figure 7.

These instructions all have a one in bit 27 and a zero in bit 26.

condition	1	0	opcode	dest reg	base reg	0	displacement
31 28	27	26	23	22 19	18 15	14	13 0

Figure 7: The type 1 format: load and store instructions.

3.3.1 Register Indirect with Immediate Displacement Mode

In this mode (e.g. `[r1,#4]`) a base register and immediate value form the address of the location to be read or written. The base register holds a 24-bit value and the immediate offset is a 14-bit signed offset. These two values are added to form the effective address of the target memory location. This mode is typically used to access fields in structures referenced by the base register, or to access values in a stack frame.

The mode can be identified by a zero in bit 14.

Examples:

```
str     a1,[sp,#a]     ; store argument 1 (r0) at fixed offset a in frame
ldr     r0,[sp,#a]     ; recover previously stored value
```

condition				0	opcode				dest reg				src reg				0	exponent					value								
condition				0	opcode				dest reg				src reg				1	0	0	shop		src reg 2				shift count					
condition				1	0	opcode			dest reg				base reg				0	signed offset from base register													
31			28	27	26	25	24	23	22			19	18			15	14	13	12	11	10	9	8		6	5	4	3			0

Figure 8: Format alignments.

Encoding					As'y	
27	26	25	24	23	Code	Instruction
0	0	0	0	0	add	add
0	0	0	1	0	sub	subtract
0	0	1	0	1	orr	bitwise or
0	0	1	1	0	and	bitwise and
0	1	0	1	1	mov	move
0	1	1	0	0	mvn	move inverted
0	1	1	0	1	swi	software interrupt
1	0	0	0	0	ldr	load register
1	0	0	0	1	str	store register
1	0	1	0	0	adr	form address

Figure 9: Encoding of operation codes.

Notes: The number of leftmost 1's in opcode (0, 1, or 2) dictates instruction format. The least significant bit of branch instructions is actually the sign bit of the adjacent displacement field, and not formally part of the opcode.

4 The Tepid Instruction Set Architecture

What follows is a detailed description of the instructions that make up the Tepid instruction set. All of the arithmetic instructions have the option of updating the condition codes. That option is indicated by a trailing `s` on the operation code. The treatment of the condition code register described here assumes that option has been specified. If the operation code does not request an update, the condition codes are not affected.

Most operations may be conditionally executed by specifying a condition-specific suffix (see Figure 3). If the condition is met, the instruction is executed. If the condition is not met, the instruction is ignored and has no effect. If no suffix is specified, the instruction is always executed.

The encoding of the operation is detailed in Figure 3.3.1, and is also given in the description of each instruction. The format of the instruction is indicated by bit 27.

ADD—Add Values (00000, Arithmetic)

Computes the 32-bit arithmetic sum of the left and right source operands and stores the result in the destination register. The condition code bits `N`, `Z`, `C`, and `V` are set to reflect the condition of the result.

ADR—Form Address (10100, Load/Store)

Construct a 24-bit absolute address from a load/store memory specification. The condition code bits are unaffected by this operation. This instruction does not access memory.

AND—Bitwise Logical And (00110, Arithmetic)

Compute the logical and of corresponding bits of the left and right source values, and store the result in the destination register. The `N` and `Z` bits are set based on the result, and the `C` and `V` bits are cleared.

LDR—Load Register (10000, Load/Store)

The destination register is loaded from the memory location indicated by the memory reference. This instruction does not change the condition code.

MOV—Move Value (01011, Arithmetic)

This instruction places the value computed in the right hand source into the destination register. This instruction does not have a left-hand source (it is encoded as `r0`). The N and Z condition code bits, if updated, reflect the condition of the destination value. The `V` bit is cleared.

MVN—Move Complemented Value (01100, Arithmetic)

This instruction places the 1's complement of the value computed on the right, into the destination register. The instruction does not have a left-hand source (it is encoded as `r0`). The `N` and `Z` bits, if updated, reflect the condition of the destination value.

Because of the limitations of constructing immediate values with large numbers of 1 bits, this operation is typically used to load immediate values with few zeros. The programmer should be aware, however, that the source is complemented and not negated. To fully compute the 2's complement (the negation) of the source, the destination register must be incremented by one.

ORR—Bitwise Logical Or (00101, Arithmetic)

The bits of the left and right source values are or-ed. The `N` and `Z` condition code bits if updated are set to reflect the state of the result. The `V` bit is cleared.

STR—Store Register (10001, Load/Store)

The 'destination' register is written to the memory location indicated by the memory reference. This instruction does not change the condition code.

SUB—Subtract Values (00010, Arithmetic)

The right hand source is subtracted from the left and the result is then written to the destination register. The condition codes are set to reflect the condition of the subtraction.

SWI—Software Interrupt (01101, Arithmetic)

A *software interrupt* is performed. This instruction is included in the description of Tepid since you will want to use it when writing programs to run under the Warm simulator to do input and output. It will not, however, be supported by the implementation of Tepid that you create using Logisim.

This instruction does not specify a destination or a left hand source register (they are both encoded as `r0`). The value computed by the right hand source indicates a software interrupt vector number. If an argument is passed to, or if a result is returned from the interrupt, register `r0` is used for that purpose. The condition codes, if updated, reflect the final value of this register.

Vector numbers below 16 (system software interrupts) are handled by hardware. For larger vector numbers, the instruction writes the vector number to register `r0` and then performs a branch with link to memory location 8. The code at this location is responsible for executing the desired software service.

4.1 An Example Program

Figure 4.1 demonstrates the Tepid instruction set by implementing a modified form of Euclid's algorithm to compute the greatest divisor of `a` and `b`, equivalent to the following function written in C:

```
int gcd(int a, int b)
// pre: 0 <= a <= b
// post: return greatest common divisor of both a and b
{
    if (a == 0) return b;       // b divides 0 and b
    if (a > b) return gcd(b,a); // swap to meet pre-condition
    return gcd(a, b-a);          // b%a is < b
}
```

The gcd routine requires its two parameters to be stored in `r1` (`a`) and `r2` (`b`), and the result will be returned in register `r0`.

Because the procedure potentially calls another procedure (it's recursive), the link register must be saved in memory on the stack.

All the arithmetic instructions have three operands: the destination (always a register), the left hand source (also a register), and the right hand source (always the result of a shift). Branch are specified using the `adr` instruction with pc as the destination register. Where the branches are conditional, an appropriate suffix specifying the required condition code setting follows the `adr` opcode. The `subs` instructions preceding these conditional branches take the place of compare instructions. The values they place in r13 are never used. The gcd function is called using an `adr` preceded by an `add` instruction that places the address of the instruction following the `add` in the link register (`lr`).

```
;; int main( int argc, char * argv[] ){
main:
;;     int x,y;         x in r1, y in r2

;;     scanf( "%d", &x );
      swi      #SysGetNum
      mov      r1,r0

      ;;     scanf( "%d", &y );
      swi      #SysGetNum
      mov      r2,r0

      ;;     print( "%d", gcd( x, y ) );

      add      lr,pc,#2
      adr      pc,gcd
      swi      #SysPutNum

      ;; }
      swi      #SysHalt


      ;; int gcd( int a, int b) {
      ;;        a in r1, b in r2
      ;;        lr saved at offset 0 in 1 word stack frame
gcd:
      sub      sp,sp,#1
      str      lr, [sp, #0]

      ;;     if ( a == 0 ) return b;
      subs     r12,r1,#0
      adrne    pc,else1
      mov      r0,r2
      adr      pc,return

      ;;     if ( a > b ) return gcd( b, a);
else1:
      subs     r12,r1,r2
      adrle    pc,else2
      mov      r0,r2
      mov      r2,r1
      mov      r1,r0
      add      lr,pc,#2
      adr      pc,gcd
      adr      pc,return

      ;;     return gcd( a, b-a );
else2:
      sub      r2,r2,r1
      add      lr,pc,#2
      adr      pc,gcd

return:
      add      sp,sp,#1
      ldr      pc,[sp, #-1]
```

Figure 10: A machine code translation of a greatest common divisor routine.