Williams Academic Risc Machine (WARM)
Duane A. Bailey
November 15, 2011 (software version: 2.135)

This document describes the Williams Academic Risc Machine (WARM) instruction set architecture (ISA). This hypothetical machine is motivated by the design of the ARM[1] family of processors, but has (often substantially) different functionality. The WARM 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

The Williams Academic Risc Machine (WARM) is a 32-bit RISC-style virtual processor that is loosely modeled after the ARM family of architectures.[2] The WARM processor has a 32-bit datapath with a 24-bit address bus addressing 16 million words of store.

The WARM 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 WARM architecture, this deficiency is counteracted in two fundamental ways: (1) most instructions are able to shift one operand with no overhead, and (2) all WARM 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 WARM 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, WARM 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 next instruction to be executed. 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 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 the WARM makes no assumptions about the particular approach used.

[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.

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 the WARM, and their common interpretations.

N	Z	C	V	(not used)		program counter	
31	30	29	28	27	24	23	0

Figure 2: The interpretation of `r15`, or `pc`, the program counter and condition code register.

The WARM supports access to a 16 million word addressable store. As a result only the low 24-bits of a register are ever used develop an effective address. Because the most significant 4-bits of the program counter are unused, the current condition code register is saved in these bits (the CCR). These four bits, named N, Z, C, 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 carry (C) or signed overflow (V). Figure 2 documents this organization. When the `pc` is written to or from the stack using the `STM` or `LDM` instructions, the condition code bits are saved to and recalled from the stack with the `pc`. In all other `pc`-related operations, only the low 24 bits are manipulated.

3 Instruction Formats

The instructions of the machine are encoded in one of 3 basic formats: type 0 (or *arithmetic*), type 1 (or *load/store*), and type 2 (or *branch*).

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.

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`≠`V`)
1	0	1	–	`le`	less or equal ((`Z`= 1) or (`N`≠`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. The set condition code bit is always high for `cmp` and `tst` instructions.

Figure 3: The condition encodings.

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.

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.

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 wide variety of shift operations 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 (four subformats)														
31			28	27	26			23	22			19	18			15	14														0

Figure 4: The type 0 format: arithmetic instructions.

3.2.1 Immediate Mode

Immediate mode (limited to use on as the right-hand source) allows for the insertion of values into a computation that are derived from the instruction itself. 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

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

Figure 5: The second arithmetic source formats.

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`). The type of shift may be a logical or arithmetic shift in either direction (`lsl` or `asl`, `lsr`, or `asr`), or a right rotation (`ror`). If the shifting operation would set the condition code bits, the carry bit (`C`) is always the last bit shifted out of the register, while the `Z` (zero result) and `N` (negative result) bits are set based on the resulting bits of the shifted register value. 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 ; C holds the sign
add     r0,r1,r2          ; the r2 is encoded as r2, lsl #0
```

3.2.3 Register Shifted by Register Mode

In this mode, the right hand source register is shifted by an unsigned immediate value found in a second register (for example, `r1, lsr r2`). The value is unsigned, and large values will typically clear the register. The type of shift may be a logical or arithmetic shift in either direction (`lsl` or `asl`, `lsr`, or `asr`), or a right rotation (`ror`). If the shifting operation would set the condition code bits, the carry bit (`C`) is always

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.

the last bit shifted out of the register, while the `Z` (zero result) and `N` (negative result) bits are set based on the resulting bits of the shifted register value.

This mode is identified by the the value 101 in bits 14 through 12.

Examples:

```
mul     r0, r1, r2, asl r3  ; r0 is r1*r2*(1<<r3)
movs    r9, r9, lsr r8      ; equivalent to Intel's shr r8,r9
```

3.2.4 The Register Product Mode

This mode causes signed values in two registers (for example `r2, r3`) to be multiplied to determine the right hand source value. Its use is limited to the fused multiply add instruction (`mla`). In that instruction the left source is added to the product found on the right and stored in the destination. The use of this mode with other instructions is undefined.

This mode is identified by the the value 110 in bits 14 through 12.

Examples:

```
mla    r0,r1,r2,r3          ; r0 is assigned r1+(r2*r3)
```

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[3] reflect the typical ways that registers are transferred to memory. The format of these instructions is detailed in Figures 7 and 8.

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

condition	1	0	opcode	dest reg	base reg	displacement (two subformats)
31 28	27	26	23	22 19	18 15	14 0

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

[3]The load and store multiple instructions (`ldm` and `stm`) are actually encoded as arithmetic instructions.

0	signed offset from base register													
1	0	0	shop		index reg				shift count					
14	13	12	11	10	9			6	5					0

Figure 8: The displacement subformats of 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
```

3.3.2 Register Indirect with Indexing Mode

In this mode (e.g. `[r1, r2, lsr #2]`) a base register and the displacement computed from a shifted register form the address of the location to be read or written. The base register holds a 24-bit value and the shifted register is a 24-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 arrays.

The displacement is encoded in exactly the same manner as in Register Shifted Immediate Mode (Section 3.2.2). The mode can be identified by the pattern 100 in bits 14 through 12.

Examples:

```
adr     r0, [r1, r2, lsl #2]     ; form pointer to element r2 in array r1 (item size=4)
```

3.4 Type 2: Branch Format

Instructions that branch are essentially instructions that perform a move of a pointer into the program counter (`pc`) register. Because the construction of full pointers can be difficult and because 1 of 6 instructions in a typical program is a branch, a special mode is reserved for that purpose. Branches compute their destination by adding a 24-bit signed displacement (directly encoded in the low 24 bits of the instruction) to the address of the instruction (ie. the program counter). The encoding of these instructions is detailed in Figure 10.

These instructions all have ones in bits 27 and 26. Be aware that the sign bit of the displacement is found in bit 23. This bit, in other instructions, is part of the operation code.

Examples:

```
b     loop    ; branch to loop; the assembler computes the offset
beq   loop    ; branch to loop if the last condition setting instruction generated zero
```

condition	1	1	opcode	displacement (signed offset from instruction address)
31 28	27	26	25 24	23 0

Figure 9: The type 2 format: branching and PC-relative instructions.

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

Figure 10: Format alignments.

Encoding					As'y		Encoding					As'y	
27	26	25	24	23	Code	Instruction	27	26	25	24	23	Code	Meaning
0	0	0	0	0	`add`	add	1	0	0	0	0	`ldr`	load register
0	0	0	0	1	`adc`	add with carry	1	0	0	0	1	`str`	store register
0	0	0	1	0	`sub`	subtract	1	0	0	1	0	`ldu`	load/pop register
0	0	0	1	1	`cmp`	compare	1	0	0	1	1	`stu`	store/push register
0	0	1	0	0	`eor`	exclusive or	1	0	1	0	0	`adr`	form address
0	0	1	0	1	`orr`	bitwise or	1	0	1	0	1		
0	0	1	1	0	`and`	bitwise and	1	0	1	1	0		
0	0	1	1	1	`tst`	test bits	1	0	1	1	1		
0	1	0	0	0	`mul`	multiply	1	1	0	0	0	`b`	branch (forward)
0	1	0	0	1	`mla`	fused multiply add	1	1	0	0	1	`b`	branch (backward)
0	1	0	1	0	`div`	divide	1	1	0	1	0	`bl`	branch and link (forward)
0	1	0	1	1	`mov`	move	1	1	0	1	1	`bl`	branch and link (backward)
0	1	1	0	0	`mvn`	move inverted	1	1	1	0	0		
0	1	1	0	1	`swi`	software interrupt	1	1	1	0	1		
0	1	1	1	0	`ldm`	load multiple	1	1	1	1	0		
0	1	1	1	1	`stm`	store multiple	1	1	1	1	1		

Figure 11: 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 WARM Instruction Set Architecture

What follows is a detailed description of the instructions that make up the WARM instruction set. Unless otherwise specified, all of these 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 11, and is also given in the description of each instruction. The format of the instruction is indicated by bit 27 and, if it is 1, bit 26.

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.

ADC—Addition with Carry In (00001, Arithmetic)

Computes the 32-bit arithmetic sum of the carry bit, and the left and right source operands, storing 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.

B—Branch (1100-, Branch)

Jump to an arbitrary location by loading the program counter with the destination address. The condition code bits are unaffected.

BL—Branch and Link (1101-, Branch)

Jump to an arbitrary location by loading the program counter (`pc`) with the address and loading the link register (`lr`) with the address of the instruction following this branch. The condition code bits are unaffected.

This instruction is typically used to provide support for calling functions. The `bl` serves as a method-calling instruction and the routine returns by moving the link register into the program counter.

CMP—Compare Values (00011, Arithmetic)

The right hand source is subtracted from the left, the condition codes are set to reflect the result, and the result is then discarded. This instruction does not have a destination (it appears as `r0`) and the `s` suffix is always implied.

DIV—Signed Divide (01010, Arithmetic)

The left hand source is divided by the right. The `N` and `Z` bits are set based on the condition of the result. The `C` and `V` bits are cleared. If the divisor is zero, the instruction halts the program.

EOR—Bitwise Exclusive Or (00100, Arithmetic)

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

LDM—Load Multiple Registers (01110, Arithmetic)

This instruction loads multiple registers from a location pointed to by the destination register. The right hand operand specifies a 16-bit mask that identifies the registers to be loaded. The bits 0 through 15 in this mask, if set, indicate the respective register is to be popped. Lower numbered registers are always retrieved from lower addresses. The destination register (if not, itself, popped) is incremented n times if n registers are to be loaded. When the register is, itself, loaded, the operation proceeds as described, but the final value of the destination register is the value restored from the stack. If the `pc` is one of the registers involved, the condition codes are set from the value saved to the `pc`. Otherwise, the condition codes are unaffected by this operation.

LDR—Load Register (10000, Load/Store)

The destination register is loaded from the memory location indicated by the memory reference. The `N` and `Z` bits of the condition code register, if updated, reflect the value moved. The `C` and `V` bits are cleared.

LDU—Load and Update Register (10010, Load/Store)

The destination register is loaded from a location specified by a base and offset reference. If the offset is negative, the offset is added to the base register to form the effective address. The value at that address is moved to the destination register, and the effective address is written to the base register.

If the offset is positive, the base register, alone, forms the effective address. The value at that address is moved to the destination register. The offset is then added to the base register value and the sum is written back to the base register. The condition codes, if set, reflect the condition of the value stored in the destination register. The `C` and `V` bits are cleared.

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 `C` and `V` bits are 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.

MUL—Signed Multiply (01000, Arithmetic)

This instruction computes the signed 32-bit product of the left hand and right hand 32-bit sources and the result is written to the destination. The condition codes, if updated, reflect the condition of the result. The `C` and `V` bits are cleared.

MLA—Fused Multiply Add (01001, Arithmetic)

This instruction has one encoding that specifies four registers. The right hand source is the product of the two registers specified using the Register Product Mode. The left and right sources are then added together and stored in the destination. The condition code bits are updated to reflect the final addition.

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 `C` and `V` bits are cleared.

STM—Store Multiple Registers (01111, Arithmetic)

This instruction stores multiple registers to a location pointed to by the destination register. The right hand operand specifies a 16-bit mask that identifies the registers to be loaded. The bits 0 through 15 in this mask, if set, indicate the respective register is to be pushed. Lower numbered registers are always stored at lower addresses. The destination register is decremented n times if n registers are to be stored. When the register is, itself, stored, the operations proceeds as described, but the final value of the destination register is the value written to memory. If the program counter `pc` is stored to the stack, the condition code bits are saved as the top four bits. The condition codes are unaffected by this operation.

STR—Store Register (10001, Load/Store)

The 'destination' register is written to the memory location indicated by the memory reference. The `N` and `Z` bits of the condition code register, if updated, reflect the value moved. The `C` and `V` bits are cleared.

STU—Store and Update Register (10011, Load/Store)

The destination register is written to a location specified by a base and offset reference. If the offset is negative, the offset is added to the base register to form the effective address. The destination value is written to that address, and the effective address is written to the base register.

If the offset is positive, the base register, alone, forms the effective address. The destination value is written to that address. The offset is then added to the base register value and the sum is written back to the base register. The condition codes, if set, reflect the condition of the value written to memory. The `C` and `V` bits are cleared.

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 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.

TST—Test Bits (00111, Arithmetic)

The right hand source is logically and-ed with the left, the condition codes are set to reflect the result, and the result is then discarded. This instruction does not have a destination (it appears as `r0`) and the `s` suffix is always implied.

4.1 An Example Program

Figure 12 demonstrates the Warm 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(b%a,a);          // b%a is < a
}
```

It is assumed that this procedure is 'called' using the branch/link instruction, which places the address of the next instruction after the call in the link register. This 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. Here, the `lr` register is 'pushed' on the stack with the store register and update. Because the 'offset' in the addressing mode is negative, the `sp` is modified before the store. When the register is popped off (in the instruction at pc=18), the offset is positive, which retrieves the stacked value before the stack register is updated.

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). Since the `cmp` instruction does not compute a value to be written back to the destination, the destination is not provided, but appears in the instruction as `r0`. That register specification will be ignored when the instruction is executed. Note also that the implied subtraction is computed as (left source - right source), which is different than on, say, Intel-based architectures. Note, also, that the 'set condition code' bit (bit 28) is implicitly set for both compares in this program.

The conditional branch from the instruction at pc=3 is specified with a conditional execution of a plain branch (`b`). Any instruction may be adopt these condition-specifying extensions. It is useful to note that all branching operations are pc-relative. That is, they branch a number of instructions either forward or backward. If we load this program in a different section of memory, it will continue to branch correctly.

Finally, the `mul` and `div` instructions compute 32-bit signed values. Because the `div` instruction does not compute the remainder, we must compute it explicitly.

```
             Machine Code
pc: cnd s opc'd dest lhs     rhs source/disp         Assembly            Comments
00: 000 0 10011 1110 1101    011111111111111 gcd:    stu lr, [sp, #-1] ; push lr; l/s format; will be moved to pc
01: 000 1 00011 0000 0001   00000 0000000000         cmp r1,#0         ; no dest; comput r1-#0; sets cc; #0=0*(1<<0)
02: 011 0 1100      000000000000000000000011         bne else1         ; branch to (pc=2)+(disp=3) = 5 (else1:)
03: 000 0 01011 0000 0000 100 00 0010 000000         mov r0,r2         ; lhs is ignored; rhs is r2, lsl #0
04: 000 0 1100      000000000000000000001110         b   return        ; branch to 4+14=18
05: 000 1 00011 0000 0001 100 00 0010 000000 else1:  cmp r1,r2         ; no dest; lhs is r2, rhs is r1, lsl #0
06: 101 0 1100      000000000000000000000110         ble else2         ; cond is (101 => le) checks (Z==1 or N!=V)
07: 000 0 01011 0000 0000 100 00 0010 000000         mov r0,r2         ; painful swap
08: 000 0 01011 0010 0000 100 00 0001 000000         mov r2,r1         ;              of r1
09: 000 0 01011 0001 0000 100 00 0000 000000         mov r1,r0         ;                    and r2
10: 000 0 1101      111111111111111111110110         bl  gcd           ; lr<-11, pc<-pc-10; 10-10 = 0; back branch
11: 000 0 1100      000000000000000000000111         b   return        ; note sign of displacement -> b. forward
12: 000 0 01010 0011 0010 100 00 0001 000000 else2:  div r3,r2,r1      ; no mod operator in warm! r3=(int)(r2/r1)
13: 000 0 01000 0011 0011 100 00 0001 000000         mul r3,r3,r1      ;         r3 = r2*((int)(r2/r1))
14: 000 0 00010 0011 0010 100 00 0011 000000         sub r3,r2,r3      ;         r3 = r1-(r2*((int)...)) = residue
15: 000 0 01011 0010 0000 100 00 0001 000000         mov r2,r1         ; a is second param
16: 000 0 01011 0001 0000 100 00 0011 000000         mov r1,r3         ; b%a is first param
17: 000 0 1101      111111111111111111101111         bl  gcd           ; lf<-18; pc = pc-17
18: 000 0 10010 1111 1101    000000000000001 return: ldu pc, [sp, #1]  ; to return: pop this routine's lr
```

Figure 12: A machine code translation of a greatest common divisor routine.