The Williams Instructional Demonstrator (WIND)

Duane A. Bailey

November 29, 2011 (software version 2.139)

This document describes the Williams Instructional Demonstrator, a machine with a very simple Intel-like instruction set. It features a 32-bit datapath, a 32-bit address space, and many of the most heavily used addressing modes, each of which may be used with most instructions. Other features, including support for accessing a small amount of custom logic make it useful as the host in implementing a small virtual machine emulator like the WAVE.

Introduction

The WIND is a simple machine designed with many of the instruction set and architectural features of the x86-32/64 machines developed by AMD and Intel. It was designed to demonstrate the features typical of this type of CISC design. The WIND has a 32-bit ('word') datapath and a 32-bit address space able to access a 4-billion word addressable store. All instructions are word-based operations.

Unlike the x86 architectures, the WIND has general purpose 32-bit registers, `r0` through `r15`, as well as an instruction pointer, `rip`, and a 4-bit condition code register, `ccr`. Register `r7` is the stack pointer (typically written, `rsp`). The stack pointer points the top of the data stack and is an integral part of some instructions (`push`, `pop`, `call`, and `return`). Register `r6` (written `rbp`) is general purpose, but is often used as a stable pointer into the most recent call frame.

Register	Alias	Typical Use
`r0`	–	return value; trap argument and return
`r1`	–	general purpose (callee saved)
`r2`	–	argument 4
`r3`	–	argument 3
`r4`	–	argument 2
`r5`	–	argument 1
`r6`	`rbp`	frame pointer
`r7`	`rsp`	stack pointer
`r8`	–	argument 5
`r9`	–	argument 6
`r10`	–	general purpose (callee saved)
`r11`	–	general purpose (callee saved)
`r12`	–	generap purpose (callee saved; avoid)
`r13`	–	general purpose (callee saved)
`r14`	–	general purpose (callee saved)
`r15`	–	general purpose (callee saved)
`rip`	–	program counter
`ccr`	–	condition code register (low 4 bits: nzcv)

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

The instruction pointer (`rip`) points to the instruction currently being executed. The programmer can manipulate the `rip` directly using register direct and displacement modes, but it is more typically accessed

as a side-effect of instructions like `jmp` (which stores it) and `lea` which often uses it as a means of converting `rip`-relative addresses into absolute.

The `ccr` holds the four condition code bits, `N`(negative result), `Z`(zero result), `C`(carry out), and `V`(overflow). The CCR is modified by arithmetic instructions, most logical instructions, all shifting instructions, and tests. It may be accessed directly by move instructions and an indirectly in the conditional branch and move instructions.

The WIND Operand Addressing Modes

The wind supports the following operand modes, which will be familiar to programmers who have used the x86 instruction sets:

Immediate Introduces a constant—written `$13`—in an instruction. Because the value has no effective address and is not associated with a register, it cannot be written to. It can, however, be used on the right side of the `cmp` instruction, which does not save its result.

Register Direct The desired value is found in a register, written, for example as `r0`. All general purpose registers and the instruction pointer are accessible using this mode.

Condition Code Direct This mode—written `ccr`—allows direct access to the 4-bit register holding the condition code bits. The mode is limited to use as the source or destination operand in the move and conditional move instructions.

Register Indirect In this mode—written `4(r3)`—the effective address is the sum of the displacement (here, `4`), and the contents of the register. The data is found in memory at the effective address. When the displacement is zero (as it frequently is in pointer manipulations), it may be omitted.

Register Indirect with Indexing This is an array-addressing mode and is written `3(r0,r4)`. The effective address is computed as the sum of the displacement (here, 3), the base register (`r0`), and the index register (`r4`). In variable width machines a scale is typically provided as a means of specifying a multiplier for the index register. The WIND has only one data width, so no scale is available. An equivalent result may be accomplished by a prior shift or multiply, of course.

Instruction Pointer Relative This mode is implied by using a label in an instruction. The effective address is the value of the label (ie. the address of a data value or instruction). The address is converted to an offset from the current instruction (ie. the address of the word holding the displacement in the instruction). The displacement is 32 bits.

The WIND Instruction Set

What follows are quick descriptions of the instructions that make up the WIND instruction set. The descriptions are meant to be helpful to programmers who have had experience writing assembly code for Intel's x86-64 instruction set. Having said that, it is important to remember that the WIND is a 32-bit architecture.

ADD—Add (0x14)

The `add` instruction adds the source into the destination. It sets the `N`, `Z`, `C`, and `V` bits to reflect the outcome of the operation.

AND—Logical And (0x10)

The `and` instruction computes the bitwise logical and of the two values and writes the result to the destination. It sets the `N` and `Z` bits to reflect the outcome of the operation. The `C` and `V` bits are cleared.

CALL—Call Routine (0x1E)

The `call` instruction pushes the address of the next instruction on the stack and then branches to the effective address of its operand. The instruction does not modify the condition code bits.

CMOVcc—Conditional Move (0x0A-0x0F)

Based on the values of the condition code bits, this instruction either moves a source value into a destination location (memory or register), or if the condition is not met, is ignored. The condition may be value less (`cmovl`), value less or equal (`cmovle`), value equal (`cmove`), value not equal (`cmovne`), value greater or equal (`cmovge`), or value greater (`cmovg`). The required condition code settings are similar to those outlined in the WARM *Architecture Manual*. This instruction does not, itself, set condition codes.

CMP—Compare (0x13)

This instruction subtracts the source from the destination and throws away the result after setting the condition code bits. It sets the `N`, `Z`, `C`, and `V` bits to reflect the outcome of the operation.

DIV—Divide (0x17)

This instruction divides the 32-bit source into the 32-bit destination and writes the 32-bit result to the destination. The remainder is not computed by this instruction. It sets the `N` and `Z` bits to reflect the outcome of the operation. The `C` and `V` are cleared. Division by zero will halt the machine.

JMP—Unconditional Branch (0x01)

This instruction allows the program to jump to an arbitrary location by loading the `rip` register with the destination address. This instruction does not set any condition codes.

Jcc—Conditional Branch (0x02-0x07)

Based on the values of the condition code bits, this instruction branches to the destination address or, if the condition is not met, continues execution with the next instruction. The condition may be value less (`jl`), value less or equal (`jle`), value equal (`je`), value not equal (`jne`), value greater or equal (`jge`), or value greater (`jg`). The required condition code settings are similar to those outlined in the WARM *Architecture Manual*. This instruction does not, itself, set any condition codes.

LEA—Load Effective Address (0x08)

This instruction moves the effective address of a (memory-referencing) source operand, and stores it in the destination. It does not read memory, and it does not change the condition codes.

MOV—Move Value (0x09)

This instruction moves the data referenced by the source to the location indicated by the destination. The move instruction does not change the condition codes.

The instruction also supports a move of a 4 bit value to or from the condition code register `ccr`. In the form where the ccr is the destination, the condition code is directly set by the instruction.

MUL—Multiply (0x16)

This instruction multiplies the source and destination and writes the result to the destination. It sets the `N` and `Z` bits to reflect the outcome of the operation. The `C` and `V` are cleared.

OR—Logical Or (0x11)

The `or` instruction computes the bitwise logical or of two values and writes the result to the destination. It sets the `N` and `Z` bits to reflect the outcome of the operation. The `C` and `V` bits are cleared.

PUSH—Push Value on Stack (0x1C)

The `push` instruction decrements the stack pointer (`sp`) and writes its operand value to that location in memory. The instruction does not modify the condition code bits.

POP—Retrieve Value from the Stack (0x1D)

The `pop` instruction reads the value in the memory location pointed to by the stack pointer (`sp`), stores in the location indicated by the operand, and then increments the stack pointer. The instruction does not modify the condition code bits.

RET—Return from Routine (0x1F)

The `ret` instruction takes no operands. The instruction pops the instruction pointer from the stop of the stack, incrementing the stack pointer. The instruction does not modify the condition code bits.

SAR—Arithmetic Shift Right (0x1A)

This instruction shifts the destination to the right by the number of bits indicated by the source, filling with copies of the sign bit. It writes the result to the destination. It sets the `N` and `Z` bits to reflect the outcome of the operation. The `C` reflects the value of the last bit shifted out. The `V` bit is cleared.

SHL—Logical/Arithmetic Shift Left (0x18)

This instruction shifts the destination to the left by the number of bits indicated by the source, always filling with zeros. It writes the result to the destination. It sets the `N` and `Z` bits to reflect the outcome of the operation. The `C` reflects the value of the last bit shifted out. The `V` bit is cleared.

SHR—Logical Shift Right (0x19)

This instruction shifts the destination to the right by the number of bits indicated by the source, filling with zeros. It writes the result to the destination. It sets the `N` and `Z` bits to reflect the outcome of the operation. The `C` reflects the value of the last bit shifted out. The `V` bit is cleared.

SUB—Subtract (0x15)

This instruction subtracts the source from the destination and writes the result to the destination. It sets the `N`, `Z`, `C`, and `V` bits to reflect the outcome of the operation.

TEST—Bit test (0x13)

This instruction performs a bitwise logical and of the source and destination and throws away the result after setting the condition code bits. It sets the `N`, `Z`, `C`, and `V` bits to reflect the outcome of the operation.

TRAP—Software Trap (0x00)

This instruction performs a software trap whose vector number is provided as the operand. If the vector number is below 16, a system service is provided. These services are precisely those services documented in the *WARM Assembler and Interpreter Guide* and include reading and writing characters and numbers, generating random numbers, loading overlays, etc. Most of these services read a parameter from register `r0` and/or leave a result in `r0`. No condition codes are modified by this instruction.

XOR—Logical Exclusive Or (0x12)

The `xor` instruction computes the bitwise logical exclusive or of two values and writes the result to the destination. It sets the `N` and `Z` bits to reflect the outcome of the operation. The `C` and `V` bits are cleared.

The WIND Instruction Format

Because the WIND is an Intel-like machine with few constraints on the operands to an instruction, its instruction format has variable length. This sectdion describes the current layout of the WIND instructions, but the format may change (for example, to make use of reserved bits) at any time in the future.

Instructions are encoded by one, two, or three 32-bit words, depending on the complexity of the constants involved in the encoded operation. The first or *introductory* word of the instruction encodes the opcode and zero, one, or two operands, depending on the operation involved.

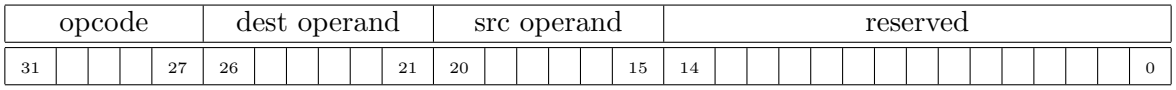

Figure 2: Introductory word of all WIND instructions.

Condition						Meaning	Syntax	Extension
26/20	25/19	24/18	23/17	22/16	21/15			
0	0	register				register direct	`r0`	–
0	1	register				register indirect	`3(r0)`	32-bit signed disp.
1	0	base register (`r0` at right)				register indexed	`3(r0,r1)`	s. disp. (bits 4-31)/ind. reg. (bits 0-3)
1	1	0	0	0	0	immediate	`#3`	32-bit value
1	1	0	0	0	1	absolute	`3`	32-bit address
1	1	0	0	1	0	`rip` direct	`rip`	–
1	1	0	0	1	1	`rip` indirect	`label`	32-bit signed disp.
1	1	0	1	0	0	`ccr` direct	`ccr`	–

Figure 3: The condition encodings.

Encoding					As'y	Op		Encoding					As'y	Op	
31	30	29	27	27	Code	#	Instruction	31	30	29	28	27	Code	#	Meaning
0	0	0	0	0	`trap`	1	system trap	1	0	0	0	0	`and`	2	bitwise and
0	0	0	0	1	`jmp`	1	unconditional branch	1	0	0	0	1	`or`	2	bitwise or
0	0	0	1	0	`jl`	1	branch if lt	1	0	0	1	0	`xor`	2	bitwise exclusive or
0	0	0	1	1	`jle`	1	branch if le	1	0	0	1	1	`cmp`	2	compare
0	0	1	0	0	`je`	1	branch if eq	1	0	1	0	0	`add`	2	add
0	0	1	0	1	`jne`	1	branch if ne	1	0	1	0	1	`sub`	2	subtract
0	0	1	1	0	`jge`	1	branch if ge	1	0	1	1	0	`mul`	2	signed multiply
0	0	1	1	1	`jg`	1	branch if gt	1	0	1	1	1	`div`	2	signed divide
0	1	0	0	0	`lea`	2	load effective address	1	1	0	0	0	`shl`	2	shift logical left
0	1	0	0	1	`mov`	2	move value	1	1	0	0	1	`shr`	2	shift logical right
0	1	0	1	0	`cmovl`	2	conditional move lt	1	1	0	1	0	`sar`	2	shift arithmetic right
0	1	0	1	1	`cmovle`	2	conditional move le	1	1	0	1	1	`test`	2	test bits
0	1	1	0	0	`cmove`	2	conditional move eq	1	1	1	0	0	`push`	1	push value on stack
0	1	1	0	1	`cmovne`	2	conditional move ne	1	1	1	0	1	`pop`	1	pop value from stack
0	1	1	1	0	`cmovge`	2	conditional move ge	1	1	1	1	0	`call`	1	call routine
0	1	1	1	1	`cmovg`	2	conditional move gt	1	1	1	1	1	`ret`	0	return from routine

Figure 4: Encoding of operation codes.

```
pc: intro     ext 1    ext 2   | Assembly code
00:                            |         .equ    a,-1
00:                            |         .equ    b,-2
00: e0030000                   | gcd:     push    rbp
01: 48c38000                   |          mov     rsp,rbp
02: a8f80000 00000002          |          sub     $2, rsp
04: 4ac08000 ffffffff          |          mov     r1,a(rbp)
06: 4ac10000 fffffffe          |          mov     r2,b(rbp)
08: 9ad80000 00000000 ffffffff |          cmp     $0,a(rbp)
11: 28198000 00000005          |          jne     else1
13: 480b0000 fffffffe          |          mov     b(rbp),r0
15: 08198000 0000001b          |          jmp     return
17: 9acb0000 fffffffe ffffffff | else1:   cmp     b(rbp),a(rbp)
20: 18198000 00000009          |          jle     else2
22: 484b0000 ffffffff          |          mov     a(rbp),r2
24: 482b0000 fffffffe          |          mov     b(rbp),r1
26: f0198000 ffffffe5          |          call    gcd
28: 08198000 0000000e          |          jmp     return
30: 486b0000 fffffffe          | else2:   mov     b(rbp),r3
32: b86b0000 ffffffff          |          div     a(rbp),r3
34: b06b0000 ffffffff          |          mul     a(rbp),r3
36: 482b0000 fffffffe          |          mov     b(rbp),r1
38: a8218000                   |          sub     r3,r1
39: 484b0000 ffffffff          |          mov     a(rbp),r2
41: f0198000 ffffffd6          |          call    gcd
```

```
43: 48e30000                | return: mov     rbp,rsp
44: e8030000                |         pop     rbp
45: f8000000                |         ret
```